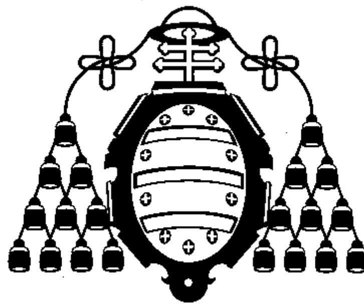
UNIVERSIDAD DE OVIEDO

Departamento de Informática



TESIS DOCTORAL

Sistema de Computación para un Sistema Operativo Orientado a Objetos basado en una Máquina Abstracta Reflectiva Orientada a Objetos

Presentada por

Lourdes Tajes Martínez

para la obtención del título de Doctora en Informática

Dirigida por el

Profesor Doctor D. Juan Manuel Cueva Lovelle

Oviedo, en un futuro no muy lejano

TABLA DE CONTENIDOS

PRESENTACIÓN DE LA TESIS

1 DISTRIBUCIÓN DE LOS CONTENIDOS
1.1 Antecedentes
1.1.1 Interoperabilidad
1.1.2 Integración de la Computación en el entorno OO
1.1.3 Sistemas Relevantes
1.2 Solución
1.2.1 SIOO: Máquina Abstracta Orientada a Objetos + Sistema Operativo Orientado a Objetos
1.2.2 Entorno de Computación Orientado a Objetos en el SIOO
1.2.3 Modelo de Objetos para la Computación
1.2.4 Integración de SOOO y MAOO: Reflectividad
1.2.5 MAOO
1.2.6 SOOO
1.3 Prototipo
1.4 Conclusiones

CAPÍTULO I LOS SISTEMAS OPERATIVOS Y EL PARADIGMA DE LA ORIENTACIÓN A OBJETOS

1 EL PARADIGMA DE ORIENTACIÓN A OBJETOS COMO TECNOLOGÍA DE DESARROLLO
1.1 Modelo de Objetos Estándar
1.1.1 Encapsulación
1.1.2 Herencia
1.1.3 Polimorfismo
1.2 Razones para migrar a un entorno OO
2 SISTEMAS OPERATIVOS ORIENTADOS A OBJETO: UNA PLATAFORMA NECESARIA
2.1 Integración no Completa del paradigma de la OO
2.1.1 Adopción Parcial
2.1.2 Adopción No Uniforme
2.2 Problemas de la adopción No completa del paradigma OO
2.2.1 Problema de la desadaptación de impedancias o salto semántico
2.2.2 La interoperabilidad entre distintos modelos de objetos
2.3 Soluciones Parciales a los problemas de interoperabilidad
2.3.1 Corba
2.3.2 SOM
2.3.3 DSOM. Distributed SOM
2.3.4 COM
2.3.5 Ventajas de la solución
2.3.6 Inconvenientes
2.4 SO + OO: Una buena pareja
3 SO OO: PRIMER OBJETIVO
3.1 Ventajas de la construcción de un SOOO para solucionar los problemas de desadaptación e interoperabilidad
4 COMPUTACIÓN EN SISTEMAS ORIENTADOS A OBJETOS

CAPÍTULO II TECNOLOGÍAS PARA LA CONSTRUCCIÓN DE UN SISTEMA OPERATIVO

1 MICRONÚCLEOS
1.1 Organización del Sistema Operativo
1.2 Definición y Funciones
1.3 Ventajas e Inconvenientes de los micronúcleos
1.3.1 Ventajas
1.3.2 Limitaciones de los micronúcleos
2 SISTEMAS OPERATIVOS ESPECÍFICOS DE LAS APLICACIONES
2.1 Organización
2.2 Ventajas e Inconvenientes
2.2.1 Ventajas
2.2.2 Inconvenientes
3 FAMILIAS DE PROGRAMAS
3.1 Definición
3.2 Organización
3.2.1 Diseño guiado por la aplicación

3.2.2 Orientación a Objetos
3.3 Ventajas e Inconvenientes
3.3.1 Ventajas
3.3.2 Inconvenientes
4 ORIENTACIÓN A OBJETOS
4.1 Definición
4.2 Aproximaciones Orientadas a Objeto en los Sistemas Operativos
4.3 Ventajas de la estructuración con objetos del sistema.
4.3.1 Reutilización.
4.3.2 Portabilidad
4.3.3 Separación Interfaz/Implantación
4.3.4 Adaptabilidad
4.3.5 Facilidad para evolucionar
4.3.6 Optimización Estructurada
5 REFLECTIVIDAD: IMPLANTACIÓN ABIERTA
5.1 Necesidad de una Implantación Abierta: Dilemas, Conflictos y Decisiones
5.2 El Modelo de Implantación Abierta
5.2.1 Interfaz base versus Meta Interfaz
5.2.2 Ventajas de la división nivel base / meta nivel
5.3 Sistemas Reflectivos
5.4 Ventajas e Inconvenientes de estos sistemas
5.4.1 Ventajas
5.4.2 Inconvenientes
6 OO + REFLECTIVIDAD: UNA COMBINACIÓN PROMETEDORA

CAPÍTULO III ENTORNOS DE COMPUTACIÓN DEFINIDOS POR LOS SISTEMAS OPERATIVOS

1 NECESIDAD DE UN ENTORNO DE COMPUTACIÓN ADECUADO PARA UN SISTEMA ORIENTADO A OBJETOS
1.1 El entorno de computación como aislante del hardware
1.1.1 El entorno de computación como proveedor de abstracciones
1.2 El sistema operativo como gestor de recursos
2 REQUISITOS DE DISEÑO DE UN ENTORNO DE COMPUTACIÓN
2.1 Requerimientos funcionales
2.1.1 Abstracciones adecuadas
2.1.2 Concurrencia en el Modelo
2.1.3 Comunicación y sincronización
2.1.4 Planificación
2.2 Requerimientos no funcionales
2.2.1 Abstracción única y sencilla
2.2.2 Flexibilidad del entorno
2.2.3 Eficiencia
3 MODELO DE PROCESOS TRADICIONAL
3.1 Abstracciones
3.2 Concurrencia
3.3 Comunicación entre procesos
3.4 Planificación de Procesos
3.5 Interfaz del sistema operativo
3.6 Ventajas e Inconvenientes del modelo de procesos
3.6.1 Ventajas
3.6.2 Inconvenientes
3.7 Integración de la OO con el Modelo de Procesos
3.7.1 Un Proceso contiene un objeto.
3.7.2 Un proceso encapsula múltiples objetos.
4 MODELO DE TAREAS E HILOS
1.1 Comunicación entre hilos
1.2 Posibles Modelos de Hilos
1.2.1 Procesos ligeros
1.2.2 Hilos de Usuario
1.2.3 Scheduler Activations
1.3 Integración de la OO con el Modelo de Tareas/Hilos
2 PROBLEMAS DE LOS MECANISMOS DE COMPUTACIÓN TRADICIONALES
2.1 Pobre adecuación para el soporte de Objetos

2.2 *Demasiadas Abstracciones*
2.3 *Falta de Uniformidad*
2.4 *Inflexibilidad*
3 MODELO OBJETO/HILO
3.1 *Objetos e Hilos*
3.2 *Ventajas e Inconvenientes del Modelo*

CAPÍTULO IV VISIÓN GENERAL DE ALGUNOS SISTEMAS OPERATIVOS RELEVANTES

1 CHOICES: UN SISTEMA OPERATIVO ORIENTADO A OBJETOS BASADO EN LA REUSABILIDAD
1.1 *Arquitectura del Sistema*
1.2 *Abstracciones de Choices*
1.3 *Computación en Choices*
1.4 *Planificación*
1.5 *Manejo de Excepciones*
1.6 *Semáforos*
1.7 *Jerarquía de marcos: Reutilización del Código*
1.8 *Crítica*
1.8.1 Separación usuario / sistema: sólo flexibilidad estática
1.8.2 Uso restringido al C++
1.8.3 Falta de uniformidad en la Orientación a Objetos
1.8.4 Mecanismo de computación tradicional no Orientado a Objetos: Abundancia de Abstracciones y falta de integración con los objetos
1.8.5 Código de Sincronización en los clientes
1.9 *Características interesantes*
1.9.1 Jerarquía de clases de implementación y con interfaz OO para el usuario
2 CLOUDS: SISTEMA OPERATIVO DISTRIBUIDO
2.1 *Arquitectura del Sistema*
2.1.1 El kernel RA
2.1.2 Objetos del sistema
2.1.3 Objetos del usuario
2.2 *Computación en Clouds*
2.2.1 Modelo Objeto/Hilo
2.2.2 Control de la Concurrencia y Consistencia del estado
2.3 *Crítica*
2.3.1 Falta de adaptación
2.3.2 Modelo Objeto/Hilo
2.4 *Cuestiones Interesantes*
2.4.1 Uniformidad Objetos Usuario/Sistema
2.4.2 Ofrece la interfaz OO
2.4.3 Soporte para el control de la concurrencia
2.4.4 No está restringido a ningún lenguaje concreto
3 SPIN: MICROKERNEL EXTENSIBLE PARA SISTEMAS OPERATIVOS ESPECÍFICOS PARA LA APLICACIÓN
3.1 *Extensibilidad de grano fino*
3.2 *Mecanismos de Computación en Spin*
3.3 *Crítica*
3.3.1 Falta de uniformidad para la extensión
3.4 *Características interesantes*
3.4.1 Extensibilidad dinámica por código de usuario
3.4.2 Seguridad en la extensibilidad
3.4.3 Eficiencia
4 AEGIS: APROXIMACIÓN A UN SISTEMA OPERATIVO EXTENSIBLE
4.1 *Arquitectura del sistema*
4.2 *Abstracción para la Computación en Aegis*
4.3 *Planificación en Aegis*
4.4 *Crítica*
4.4.1 Mecanismo Complejo de Bajo Nivel para implementar la planificación
4.4.2 Mecanismo de muy bajo nivel, demasiado complejo para la mayor parte de las aplicaciones
4.5 *Cuestiones Interesantes*
4.5.1 Flexibilidad
4.5.2 Eficiencia
5 APERTOS
5.1 *Crítica*

5.1.1 Complejidad de estructura
5.1.2 Falta de uniformidad por la separación espacio/meta-espacio de objetos
5.1.3 No existe mecanismo de seguridad uniforme en el sistema
5.2 Características interesantes
5.2.1 Reflectividad para la flexibilidad
6 MÁQUINA VIRTUAL JAVA
6.1 Subsistema de Carga de Clases
6.2 Runtime Data Areas
6.2.1 Área de Métodos
6.2.2 Heap
6.2.3 Pila
6.2.4 Registros
6.3 Execution Engine en Java
6.3.1 Planificación
6.3.2 Sincronización
6.4 Características Interesantes
6.4.1 Independencia de la plataforma
6.4.2 Soporte Básico a los objetos
6.4.3 Soporte básico a la Computación
6.4.4 Máquina Multihilo
6.5 Crítica
6.5.1 Falta de Definición de una Relación clara y estable entre los objetos y el entorno de ejecución de la máquina

CAPÍTULO V UN SISTEMA INTEGRAL ORIENTADO A OBJETOS

1 SISTEMA INTEGRAL ORIENTADO A OBJETOS COMO SOLUCIÓN GLOBAL
1.1 Entorno Integral Orientado a Objetos
1.2 Soporte al Objeto en el Sistema Integral
1.3 Sistema = Mundo de Objetos
2 REQUISITOS DE DISEÑO DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS
2.1 Modelo de objetos Uniforme.
2.1.1 Abstracción y Encapsulamiento: Clases
2.1.2 Jerarquía. La relación “es-un” (herencia). La relación “es-parte-de” (agregación)
2.1.3 Modularidad
2.1.4 Tipos y polimorfismo
2.1.5 Excepciones
2.1.6 Concurrencia
2.1.7 Persistencia
2.1.8 Distribución
2.1.9 Seguridad
2.2 Homogeneidad
2.3 Simplicidad
3 ARQUITECTURA DEL SISTEMA INTEGRAL

CAPÍTULO VI EL SISTEMA OPERATIVO ORIENTADO A OBJETO: EL CEREBRO DEL SISTEMA INTEGRAL

1 EL PARADIGMA DE OO Y LOS SISTEMAS OPERATIVOS. ¿FALACIA O REALIDAD?
2 SOPORTE AL CONCEPTO DE OBJETO
2.1 Uniformidad en torno a la OO
2.1.1 Abstracción única
2.2 Modelo de Objetos único intencionadamente estándar
2.3 Abstracción Homogénea
2.3.1 Modo de trabajo exclusivamente orientado a objetos
3 ESTRUCTURACIÓN E IMPLANTACIÓN DEL SISTEMA COMO UN CONJUNTO DE OBJETOS
3.1 Diseño del Sistema Operativo como un marco OO
3.2 Sistema Operativo como conjunto de objetos en tiempo de ejecución
3.3 Interfaz oo
4 MULTILINGUAL
5 FLEXIBILIDAD
5.1 Flexibilidad: Nuevo Reto en el Diseño de Sistemas Operativos
5.2 Falta de flexibilidad. Causas y Consecuencias
5.2.1 Las Causas
5.2.2 Las Consecuencias

5.3 Distintas aproximaciones a la flexibilidad
5.3.1 Flexibilidad estática: Sistemas personalizables
5.3.2 Flexibilidad dinámica
5.4 Flexibilidad en los Sistemas Operativos
6 REFLECTIVIDAD. ARQUITECTURA PARA UN SISTEMA FLEXIBLE
6.1 OO: Necesaria, pero no suficiente
6.1.1 Necesaria ...
6.1.2 ... Pero no Suficiente
6.2 Separación Objeto – Meta-objeto
6.2.1 Meta-Objetos: Objetos
7 VENTAJAS DERIVADAS DE UN SISTEMA OPERATIVO OO
7.1 Economía de Conceptos
7.2 Rendimiento
7.3 Sistemas Personalizables
7.4 Adaptabilidad
7.5 Extensibilidad: Soporte para interfaces adaptables

CAPÍTULO VII DISEÑO DE UN ENTORNO DE COMPUTACIÓN PARA EL SISTEMA OPERATIVO OO

1 OBJETIVOS DE DISEÑO DEL SISTEMA DE COMPUTACIÓN PARA EL SOOO
1.1 Requerimientos funcionales
1.2 Requerimientos no funcionales
2 INTEGRACIÓN DE LA COMPUTACIÓN EN EL PARADIGMA DE ORIENTACIÓN A OBJETOS: UN RETO DIFÍCIL
2.1 Dificultades: Problemas de Integridad y la Anomalía de la Herencia
2.1.1 El problema de la Integridad del Objeto
2.1.2 El Problema de la Herencia
2.2 Aproximaciones para la Integración de Concurrencia y OO
2.2.1 Aproximación Ortogonal
2.2.2 Aproximación Heterogénea
2.2.3 Aproximación Homogénea

CAPÍTULO VIII REQUISITOS FUNCIONALES PARA EL SISTEMA DE COMPUTACIÓN DE UN SOOO

1 PROPORCIONAR ABSTRACCIONES ADECUADAS
1.1 Semántica del Objeto respecto a la Computación
1.2 Modelo de Objetos para la concurrencia
1.2.1 Máquina abstracta
1.2.2 Sistema Operativo
1.2.3 Máquina Abstracta + Sistema Operativo = Soporte a objetos
2 INTERACCIÓN ENTRE OBJETOS: MECANISMO DE COMUNICACIÓN UNIFORME
2.1 Modo de trabajo exclusivamente Orientado a objetos
2.2 Uniformidad en la Comunicación
2.3 Modelos de Paso de Mensajes
2.3.1 Modelo Síncrono
2.3.2 Modelo Asíncrono
2.3.3 Modelo Wait-By-Necessity
3 GESTIÓN DE LA COMPUTACIÓN CONCURRENTE
3.1 Concurrencia interna
3.1.1 Control de la Concurrencia Interna: Sincronización
3.1.2 Entorno propuesto en esta tesis
3.2 Concurrencia Externa
3.3 Sincronización
3.3.1 Sincronización en el paso de mensajes
3.3.2 Sincronización a nivel de objeto
4 PLANIFICACIÓN
4.1 Planificación interna
4.2 Planificación externa
5 MODELO DE EXCEPCIONES
5.1 Lanzar y Gestionar Excepciones
5.2 Gestión de excepciones con el paradigma de la orientación a objetos
5.2.1 Representación de las excepciones: Clases
5.2.2 Gestión de las excepciones

CAPÍTULO IX REQUISITOS NO FUNCIONALES PARA EL SISTEMA DE COMPUTACIÓN DE UN SOOO

1 MODELO ÚNICO DE OBJETOS TAMBIÉN PARA LA COMPUTACIÓN
2 ABSTRACCIÓN ÚNICA Y HOMOGÉNEA
2.1 Integración de Conceptos
2.2 Encapsulación de la Computación
2.3 Abstracción homogénea
3 VISIÓN UNIFORME DEL SISTEMA
3.1 Entorno sencillo y Economía de Conceptos
3.2 Abstracción potente
4 MECANISMO DE COMPUTACIÓN FLEXIBLE
4.1 Sistemas Tradicionales: Poco adecuados para la OO
4.2 Máquina abstracta OO + Sistema Operativo OO: Separación de política y mecanismos
4.2.1 Máquina Abstracta: Los mecanismos de bajo nivel
4.2.2 Sistema Operativo: Las políticas de Gestión
5 EFICIENCIA

CAPÍTULO X MODELO DE OBJETOS PARA EL SISTEMA DE COMPUTACIÓN

1 MODELO DE OBJETOS ACTIVO VERSUS MODELO DE OBJETOS PASIVO
1.1 Homogeneidad: Un Objetivo Irrenunciable
2 MODELO DE OBJETOS PASIVO
2.1 Objetos Pasivos
2.2 Objetos Proceso
2.3 Semántica del Modelo
2.3.1 Interacción de Objetos
3 MODELO DE OBJETOS ACTIVO
3.1.1 Objetos Activos
3.1.2 Interacción entre Objetos
3.1.3 Integración de la Concurrencia en el modelo
3.1.4 Entorno
4 COMPARACIÓN DE AMBOS MODELOS
4.1 Modelo de Objetos Pasivo
4.1.1 Ventajas
4.1.2 Inconvenientes del modelo pasivo
4.2 El modelo de objetos activo
4.2.1 Ventajas
4.2.2 Inconvenientes
4.3 Propuesta para el SIOO

CAPÍTULO XI ARQUITECTURA DEL SISTEMA INTEGRAL

1 INTEGRACIÓN DE MÁQUINA ABSTRACTA Y SISTEMA OPERATIVO PARA FORMAR UN SISTEMA INTEGRAL
2 ¿CÓMO OFRECER LA FUNCIONALIDAD DEL SISTEMA OPERATIVO?
3 IMPLEMENTACIÓN AD-HOC EN LA PROPIA MÁQUINA ABSTRACTA
3.1 Existencia de Objetos
3.2 Comunicación
3.3 Concurrencia
3.4 Sincronización
3.5 Planificación
3.6 Ventajas e Inconvenientes
3.6.1 Ventajas
3.6.2 Inconvenientes
4 ADICIÓN Y/O MODIFICACIÓN DE LAS CLASES BÁSICAS DE LA MÁQUINA CREANDO UN CONJUNTO DE OBJETOS DISTINGUIDOS O ESPECIALES
4.1 Ventajas e Inconvenientes
4.1.1 Ventajas
4.1.2 Problemas:
5 COLABORACIÓN CON EL FUNCIONAMIENTO DE LA MÁQUINA. REFLECTIVIDAD

CAPÍTULO XII ESTUDIO DE LA REFLECTIVIDAD COMO ARQUITECTURA INTEGRADORA

1 REFLEXIÓN: MODIFICACIÓN DE LA ESTRUCTURA O EL COMPORTAMIENTO DE UN SISTEMA
1.1 Reflectividad
1.2 Modelo del Sistema
1.3 Actividades básicas en un sistema reflectivo: Exposición y Reflexión
1.3.1 Exposición
1.3.2 Reflexión
2 ARQUITECTURA REFLECTIVA
2.1 Definición de una Arquitectura Reflectiva
2.2 Aplicación de la Reflectividad en sistemas Orientados a Objetos
2.2.1 Representación del modelo del sistema como un conjunto de objetos
2.2.2 Uniformidad
2.2.3 Parámetros de una arquitectura reflectiva OO
3 ESTRUCTURACIÓN DE LOS SISTEMAS REFLECTIVOS
3.1 Nivel Base y Meta-Nivel
3.2 Transferencia de control
3.2.1 Propiedades
3.2.2 Arquitecturas Reflectivas, atendiendo a la transferencia de control
3.2.3 Manipulación del entorno
4 TIPOS DE REFLECTIVIDAD
4.1 Reflectividad Estructural
4.2 Reflectividad del Comportamiento.
4.2.1 Superposición Reflectiva
4.2.2 Torre Reflectiva
4.2.3 Reflectividad del Comportamiento en sistemas OO
5 MODELOS DE REFLECTIVIDAD
5.1 Modelo MetaClase
5.1.1 Ventajas
5.1.2 Inconvenientes
5.2 Modelo MetaObjeto
5.2.1 Ventajas
5.2.2 Inconvenientes
6 La Reflectividad como Exposición de la Comunicación o Exposición de los Mensajes
6.1.1 Ventajas
6.1.2 Inconvenientes
7 META-INTERFACES Y MOP
7.1 Interfaz base versus Interfaz Meta
7.1.1 Interfaz base
7.1.2 Meta-interfaz
7.2 Meta-Object Protocols (MOPS)
8 REFLECTIVIDAD Y SISTEMAS OPERATIVOS
9 REFLECTIVIDAD EN SISTEMAS CONCURRENTES ORIENTADOS A OBJETOS
10 VENTAJAS DE UNA ARQUITECTURA REFLECTIVA APLICADA A UN SISTEMA CONCURRENTE ORIENTADO A OBJETOS

CAPÍTULO XIII SISTEMAS REFLECTIVOS

1 REFLECTIVIDAD ESTRUCTURAL. LA API REFLECTIVA DE JAVA
1.1 Modelo Reflectivo
1.1.1 Nuevas Clases: Objetos que representan el entorno en tiempo de ejecución
1.1.2 Nuevos Métodos: Conseguir la información
1.2 Componentes
2 METAJAVA
2.1 Guías de Diseño
2.2 Modelo Computacional de MetaJava
2.2.1 Separación en Nivel Base y Meta Nivel
2.2.2 Transferencia de Control
2.3 Asociación de Meta Objetos con las Entidades del Nivel Base
2.4 Estructura
2.4.1 Meta-level interface o MLI
2.5 Reflectividad del Comportamiento
1.3 Implantación de la Relación objetos base – meta objetos: Shadow Classes

2.5.1 Asociación de Meta Objetos a los Objetos del Nivel Base
2.5.2 Torre Reflectiva
2.5.3 Problemas de la Implantación
2.6 Transferencia del Nivel Base al Meta Nivel: Eventos
3 DSOM
3.1 Modelo de objetos de SOM
3.2 Reflectividad en DSOM : Metaclases
3.3 Jerarquía de Objetos SOM
3.3.1 Objeto Class SOMObject
3.3.2 Objeto Class SOMClass
3.3.3 Objeto SOMClassMgr Class y SOMClassMgrObject
3.4 Introspección de Objetos
3.4.1 Clase de un objeto
3.4.2 Clase Objeto
4 SMALLTALK
4.1 Meta Objetos en Smalltalk
4.2 Aspectos Reflectivos en Smalltalk
4.2.1 Meta Operaciones
4.2.2 Estructura
4.2.3 Envío de Mensajes
4.2.4 Control de Estado

CAPÍTULO XIV INTEGRACIÓN DE MÁQUINA ABSTRACTA + SISTEMA OPERATIVO: AÑADIR REFLECTIVIDAD AL SISTEMA

1 DISEÑO DE UN ENTORNO DE COMPUTACIÓN ORIENTADO A OBJETO
1.1.1 Requisitos Funcionales
1.1.2 Requisitos No Funcionales
1.1.3 El modelo de objetos
1.2 Entorno de Computación = Sistema Operativo + Máquina abstracta
1.2.1 Carbayonia + SO4
2 ENTORNO FLEXIBLE = REFLECTIVIDAD (SISTEMA OPERATIVO + MÁQUINA ABSTRACTA)
2.1 Reflectividad: Arquitectura Genérica
2.2 La Máquina Abstracta como Conjunto de Objetos
2.2.1 Elevación de los objetos de la Máquina abstracta OO
2.2.2 Construcción de la Máquina Abstracta Orientada a Objeto
2.3 El Sistema Operativo como Conjunto de Objetos
2.3.1 Adición del Sistema Operativo
3 TIPO DE REFLECTIVIDAD
3.1 ¿Reflectividad estructural o del comportamiento?
3.2 Reflectividad Estructural
3.3 Reflectividad del Comportamiento
3.3.1 Utilización del Paradigma de OO para describir el MetaNivel
3.3.2 Los Meta-Objetos dan Soporte a los Objetos Base
4 ARQUITECTURA REFLECTIVA PROPUESTA
4.1 Torre Reflectiva: Dos Niveles de objetos
4.2 Modelo de Reflectividad
4.2.1 Elección del Modelo
4.2.2 El Modelo de Meta-Objetos
4.3 El Nivel Base
4.3.1 Semántica de los objetos del Nivel Base
4.4 El Meta-Nivel
4.4.1 Aspectos de la máquina que migran al meta-nivel
4.4.2 El Meta Objeto
4.4.3 Representación del objeto base en tiempo de ejecución
4.5 Transferencia de Control al Meta-Nivel
4.5.1 Reflectividad Implícita: Invocación de Métodos → Meta-Computación
4.5.2 Reflectividad Explícita
4.5.3 Transferencia de Control Uniforme
4.5.4 Una Arquitectura Resultado de la composición de objetos
5 OBJETIVOS DEL META-NIVEL: SELECCIÓN DE META-OBJETOS
5.1 Objetivos del Meta-Nivel
5.2 Meta-Objetos Mensajero
5.3 Meta-Objeto Sincronizador

5.4 Meta-Objeto Planificador
5.5 Modelo de objetos implantado en el Meta-Espacio
5.6 Funcionamiento Global del nivel-base y el meta-nivel

CAPÍTULO XV LA MÁQUINA ABSTRACTA. ENTORNO DE COMPUTACIÓN BÁSICO

1 ARQUITECTURA DE REFERENCIA DE UNA MÁQUINA ABSTRACTA
1.1 Propiedades fundamentales de una máquina abstracta para un SIOO
1.2 Estructura de referencia
1.3 Juego de instrucciones
1.3.1 Instrucciones declarativas
1.3.2 Instrucciones de comportamiento
2 SOPORTE BÁSICO AL MODELO DE OBJETOS
2.1 Modelo básico de Objetos
2.1.1 Heredero de las Metodologías OO
2.1.2 Declaración de clases en la arquitectura propuesta
2.2 Clase Básica Objeto
2.2.1 Nivel de Abstracción Único: Uniformidad en torno al objeto
2.2.2 Identificador único de objetos
2.2.3 Reflectividad: Aumento del Soporte al Objeto por parte de la máquina abstracta
2.3 Polimorfismo
2.4 Excepciones
3 SOPORTE BÁSICO A LA CONCURRENCIA
3.1 Objetos Activos como Modelo de Unificación
3.1.1 Ausencia de otras Abstracciones que no sean el objeto
3.1.2 Estrecha relación, implementada a bajo nivel en la máquina abstracta, entre un objeto y su computación
3.2 Contexto de Ejecución de los Objetos
3.2.1 Representación en Tiempo de Ejecución de los Objetos por parte de la Máquina Abstracta
3.2.2 Descripción de la ejecución de un método
3.2.3 Creación y Destrucción de Hilos
3.2.4 Relación objeto-hilos
3.3 Concurrencia en el Modelo de Objetos soportado por la Máquina Abstracta
3.3.1 Introducir la concurrencia en el Sistema
3.3.2 Mecanismos de Gestión de la Concurrencia
4 COMUNICACIÓN ENTRE OBJETOS
4.1 Instrucción de la máquina
4.2 Modelos de Comunicación
4.2.1 Modelo Síncrono: Call
4.2.2 Modelo Wait-By-Necessity: Send
4.3 Reflectividad del Comportamiento
4.3.1 Paso de control al meta-nivel
4.3.2 Envío de mensajes: Ejecución de la Instrucción Call por parte de un objeto
4.3.3 Envío de mensajes: Ejecución de la Instrucción Send por parte de un objeto
5 MECANISMO DE EXCEPCIONES
5.1 Las instrucciones
5.2 Instalación de un Manejador para la excepción
5.3 Ejecución del Manejador para gestionar la excepción producida
5.3.1 El Método Actual define un Gestor en su código
5.3.2 El Método Actual no define un Gestor en su código
5.4 Gestión de Excepciones e Invocación de Métodos Síncrona
5.5 Gestión de Excepciones e Invocación de Métodos Wait-By-Necessity
5.6 Ventaja de la incorporación de excepciones
6 CONTROL DE LA CONCURRENCIA
6.1 Soporte de la Máquina Abstracta para el Control de la Concurrencia Entre Objetos
6.1.1 Paso de mensajes
6.1.2 Objetos tipo cerradura
6.2 Soporte de la Máquina Abstracta para el Control de la Concurrencia Interna de los objetos
6.2.1 Cooperación con el Sistema Operativo
6.2.2 Arquitectura Reflectiva: Cesión de Control al Sistema Operativo
7 COLABORACIÓN DE LA MÁQUINA ABSTRACTA EN EL MECANISMO DE PLANIFICACIÓN
7.1 Ciclo de Ejecución de la Máquina Abstracta
7.2 Interrupciones de Reloj
7.3 Cambio de Contexto

7.4 Planificador del Sistema.
7.4.1 El planificador de la máquina
7.4.2 Planificación a bajo nivel
8 ARQUITECTURA REFLECTIVA: ADECUACIÓN DE LA MÁQUINA ABSTRACTA PARA PERMITIR SU EXTENSIÓN POR EL SISTEMA OPERATIVO
8.1 Exponer elementos de la arquitectura de la máquina.
8.1.1 Área de Clases
8.1.2 Área de Instancias
8.1.3 Área de Referencias
8.1.4 Área de hilos.
8.2 Exponer elementos estructurales de la implantación en tiempo de ejecución de los objetos
8.2.1 Las Clases
8.2.2 Las instancias
8.2.3 Los métodos
8.2.4 Las Instrucciones
8.3 Exponer la Engine en tiempo de ejecución
8.3.1 Área de Ejecución del Objeto
8.3.2 Hilos de Ejecución
8.4 Exponer el MetaEspacio
8.4.1 El MetaEspacio de un Objeto
8.4.2 Los metaobjetos
9 VENTAJAS DEL USO DE UNA MÁQUINA ABSTRACTA
9.1 Portabilidad y heterogeneidad
9.2 Facilidad de comprensión
9.3 Facilidad de desarrollo
9.4 Compiladores de lenguajes
9.5 Implementación de la máquina
9.6 Esfuerzo de desarrollo reducido
9.7 Rapidez de desarrollo
9.8 Facilidad de experimentación
9.9 Buena plataforma de investigación
10 MINIMIZACIÓN DEL PROBLEMA DEL RENDIMIENTO DE LAS MÁQUINAS ABSTRACTAS
10.1 Compromiso entre velocidad y conveniencia aceptado por los usuarios
10.2 Mejoras en el rendimiento
10.2.1 Mejoras en el hardware
10.2.2 Optimizaciones en la implementación de las máquinas. Compilación dinámica (justo a tiempo)
10.2.3 Implementación en hardware
10.3 Resumen

CAPÍTULO XVI SISTEMA OPERATIVO: EL METAESPACIO

1 EL MUNDO DE OBJETOS DEL SISTEMA OPERATIVO
2 EL SISTEMA DE COMUNICACIÓN DEL SISTEMA OPERATIVO
2.1 Reflectividad en la invocación de métodos
2.1.1 Aspectos expuestos de la Invocación de Métodos
2.1.2 Meta-Objetos para Exponer la invocación de Métodos
2.1.3 Paso de Control del nivel Base al Meta-Nivel
2.2 Meta-Objeto Mensajero-Emisor
2.2.1 Objetivo del Meta-Objeto
2.2.2 Mecanismos de Adaptación
2.2.3 El Meta-Objeto Emisor de cerca
2.2.4 Ejecución del Meta-Objeto Mensajero-Emisor
2.3 Meta-Objeto Mensajero-Receptor
2.3.1 Ejecución del Meta-Objeto Receptor
2.3.2 Objetivo del Meta-Objeto
2.3.3 Adaptación del Mecanismo de Invocación de Métodos: Análisis del Entorno de Ejecución
2.3.4 Selección del Método
2.3.5 Otras Acciones
2.3.6 El Meta-Objeto Receptor en profundidad
2.4 Información del Meta-Nivel
2.5 Ejecución de los métodos en el Meta-Nivel
2.5.1 Diálogo en el Meta-Nivel
2.6 Ventajas de la reflectividad en la invocación a métodos

3 MODELO DE OBJETOS ACTIVO EN EL METANIVEL: CONTROL DE LA CONCURRENCIA INTERNA DE LOS OBJETOS
3.1 Reflectividad en el Control de la Concurrencia
3.1.1 Objetivos
3.1.2 Aspectos Expuestos
3.1.3 Meta-Objetos para Exponer el Control Interno
3.2 El Meta-Objeto Sincronizador
3.2.1 Objetivos del Meta-Objeto
3.2.2 Análisis de Mensajes o Política de Servicio de Invocaciones
3.3 Ejecución del Meta-Objeto Sincronizador
3.3.1 Reflectividad Explícita
Reflectividad Implícita
3.3.2 Sincronización exhaustiva de Métodos
3.3.3 Ejecución del método
3.4 Ventajas de la Reflectividad en el Control de la Concurrencia dentro de los objetos
4 EL SISTEMA DE PLANIFICACIÓN DEL SISTEMA OPERATIVO
4.1 Planificación a Varios Niveles
4.2 Reflectividad en la Planificación
4.2.1 Aspectos Expuestos de la Reflectividad en la Planificación
4.2.2 Meta-Objetos para la Planificación
4.3 Jerarquía de Planificadores
4.3.1 Jerarquía de Clases de Planificación: Reutilización del Código
4.3.2 Jerarquía de Meta-Objetos Planificadores en Tiempo de Ejecución
4.4 Planificación: Jerarquía de Planificadores en tiempo de ejecución
4.4.1 Planificación de Objetos
4.4.2 Planificación de Meta-Objetos
4.5 El Meta-Objeto Planificador
4.5.1 Planificadores de bajo nivel
4.5.2 Resto de Planificadores
4.5.3 La Raíz de la Jerarquía
4.6 Transferencia de control entre los planificadores
4.6.1 Invocación de Métodos y Retorno
4.6.2 Excepciones
5 VISIÓN DINÁMICA DEL SISTEMA

CAPÍTULO XVII PROTOTIPO DE SISTEMA INTEGRAL (MÁQUINA ABSTRACTA Y SISTEMA OPERATIVO) ORIENTADO A OBJETOS

1 PRESENTACIÓN DEL SISTEMA INTEGRAL: MÁQUINA ABSTRACTA CARBAYONIA Y SISTEMA OPERATIVO
1.1 Aplicación intensiva de los Principios de la OO
1.1.1 Aplicación de los Principios de la OO en el Diseño del Sistema Integral
1.1.2 Aplicación de los Principios de OO en la implantación del Sistema Integral
1.1.3 Reflectividad
1.2 Simplicidad
1.2.1 Aplicación de los Principios de OO
1.2.2 Mecanismo de Comunicación
2 PRESENTACIÓN DE LA MÁQUINA ABSTRACTA CARBAYONIA
2.1 Diseño de la máquina como conjunto de objetos
2.1.1 Diagrama de clases general
2.2 Implantación de los elementos de la máquina: Uniformidad en la OO
2.2.1 Clases primitivas y clases de usuario
2.3 Descripción de la Jerarquía de Clases Primitivas
2.4 Descripción de los Objetos Instancias
2.5 Descripción de los Métodos
2.6 Descripción de las Instrucciones
3 ENTORNO DE COMPUTACIÓN EN TIEMPO DE EJECUCIÓN DEFINIDO POR LA MÁQUINA
3.1 Representación del entorno en tiempo de ejecución
3.1.1 Representación de los objetos en tiempo de ejecución
3.1.2 Representación de la Computación
3.1.3 Implantación
4 CICLO DE EJECUCIÓN DE INSTRUCCIONES
4.1 Ejecución de una Instrucción
4.2 Invocación de Métodos

4.2.1 Ejecución de Métodos
4.2.2 Retorno de Resultados
4.3 Excepciones
4.3.1 Implementación
5 REFLECTIVIDAD ESTRUCTURAL EN LA MÁQUINA
5.1 Descripción de la Jerarquía de Clases para la Reflectividad Estructural
6 SISTEMA OPERATIVO: REFLECTIVIDAD DEL COMPORTAMIENTO
6.1 Descripción de la Jerarquía de Clases para la Reflectividad del Comportamiento
6.1.1 Comunicación en el Sistema Operativo
6.1.2 Modelo de Objetos para la Concurrencia en el Sistema Operativo
6.1.3 Planificación en el Sistema Operativo

CAPÍTULO XVIII CONCLUSIONES DEL TRABAJO Y RESULTADOS MÁS DESTACABLES

1 BREVE RESUMEN DEL DOCUMENTO
1.1 Planteamiento del problema: Construcción de un Sistema de Soporte de Objetos
1.2 Diseño de un entorno de Computación OO
1.2.1 Diseño de un SIOO
1.2.2 Integración de la Computación en el Entorno OO
1.2.3 Reflectividad en el Sistema
1.2.4 Máquina Abstracta y Sistema Operativo por dentro
2 RESULTADOS MÁS DESTACABLES
2.1 SIOO=MAOO+SOOO. Soporte Global al objeto
2.2 Modelo de objetos Activo: Uniformidad total
2.3 Integración de MAOO y SOOO por medio de la Reflectividad. Flexibilidad
2.4 Máquina Abstracta: Base de la torre reflectiva
2.5 Sistema Operativo: Meta Nivel

CAPÍTULO XIX FUTURAS LÍNEAS DE TRABAJO MÁS PROMETEDORAS

1 REFLECTIVIDAD BAJO DEMANDA
2 DISEÑAR Y CONSTRUIR COMPILADORES DE LENGUAJES DE PROGRAMACIÓN OO
2.1 Adecuación del Sistema Reflectivo subyacente
2.2 Análisis de las categorías reflectivas
3 CONSTRUCCIÓN DE UNA JERARQUÍA DEL SISTEMA OPERATIVO MÁS COMPLETA
4 REFLECTIVIDAD GUIADA POR EVENTOS
5 INTEGRACIÓN DE OTROS ASPECTOS EN LA ARQUITECTURA REFLECTIVA PROPUESTA
6 IMPLANTACIÓN EFICIENTE DEL PROTOTIPO

ANEXO A DESCRIPCIÓN DEL LENGUAJE CARBAYÓN: JUEGO DE INSTRUCCIONES

1 INSTRUCCIONES DECLARATIVAS: DESCRIPCIÓN DE LAS CLASES
1.1 Identificación de una Clase
1.2 Relaciones entre Clases
1.2.1 Relaciones de Herencia
1.2.2 Relaciones de Agregación
1.2.3 Relaciones de Asociación
1.3 Métodos
2 DEFINICIÓN DE MÉTODOS
2.1 Cabecera de método
2.1.1 Identificación del método
2.1.2 Referencias
2.1.3 Instancias
2.2 Cuerpo del método
3 JUEGO DE INSTRUCCIONES DE LA MÁQUINA
3.1 Instrucciones Referidas a la Manipulación de Objetos
3.1.1 Creación y Destrucción de Objetos
3.1.2 Asignación de objetos a referencias
3.2 Instrucciones Relativas a la Invocación de Métodos
3.3 Control de flujo
3.3.1 Finalización de un método
3.3.2 Salto incondicional

3.3.3 Salto condicional
3.3.4 Excepciones

ANEXO B JERARQUÍA DE CLASES PRIMITIVAS DE LA MÁQUINA

1 CLASES PRIMITIVAS BÁSICAS
1.1 Clase Object
1.2 Bool
1.3 Integer
1.4 Float
1.5 String
1.6 Array
1.7 Stream
1.8 ConStream
1.9 FileStream
1.10 Clase Lock

ANEXO C JERARQUÍA DE CLASES PRIMITIVAS REFLECTIVAS DE LA MÁQUINA

1 CLASES PRIMITIVAS PARA LA REFLECTIVIDAD ESTRUCTURAL
1.1 Exposición de los elementos que forman la Arquitectura de la Máquina
1.1.1 Clase ClassArea
1.1.2 Clase InstanceArea
1.1.3 Clase ReferenceArea
1.1.4 Clase ThreadArea
1.2 Exposición de los elementos estructurales de la implantación en tiempo de ejecución de los objetos
1.2.1 Clase _Class
1.2.2 Clase Instance
1.2.3 Clase Method
1.2.4 Clase Instruction
1.3 Exposición de la engine en tiempo de ejecución
1.3.1 Clase ExecObjectArea
1.3.2 Clase Thread
1.3.3 Clase Context
2 CLASES PRIMITIVAS PARA LA REFLECTIVIDAD DEL COMPORTAMIENTO
2.1 Clase MetaSpace
2.2 La Clase MetaObjeto
2.3 Clase Emisor
2.4 Clase Receptor
2.5 Clase Sincronizador
2.6 Clase Scheduler

REFERENCIAS

Presentación de la Tesis

La tesis que aquí se presenta, describe la actuación necesaria sobre un sistema de soporte de objetos, fundamentalmente sobre las capas y mecanismos más básicas de este sistema de soporte, con el fin de diseñar un sistema de computación concurrente OO.

Este sistema de computación debe mantenerse fiel a ciertos principios de diseño básicos, como son la uniformidad en torno a la OO y la homogeneidad del modelo de objetos en todos los elementos del sistema.

En base a ello, se diseña un Sistema Integral Orientado a Objetos, que está siendo objeto de estudio e investigación por parte de varios grupos de investigación en Tecnologías Orientadas a Objetos, incluyendo el grupo de TOO del Departamento de Informática de la Universidad de Oviedo.

Con este fin, se están desarrollando distintos aspectos a integrar en un SOO, como la distribución o la seguridad, con el propósito de dotar de entidad global al sistema.

1 Distribución de los Contenidos

Los contenidos de esta tesis están lógicamente distribuidos en cuatro partes: Antecedentes, Solución, Prototipo y Conclusiones.

Cada una de ellas supone una aproximación al problema, desde la presentación del mismo hasta la construcción de un prototipo que incluye los mecanismos necesarios para solucionarlo.

1.1 Antecedentes

La sección de antecedentes de esta tesis, comprende los Capítulos I, II, III y IV. En ellos se plantea el problema general de diseño de un SOO que actúe de soporte a objetos.

Particularmente, se centra en el problema de las actuaciones a llevar a cabo sobre este Sistema de Soporte de Objetos, con el fin de integrar la computación de objetos en el entorno.

Se realiza un breve estudio de aquellos Sistemas actuales más relevantes, ya sean OO o no, que, de una forma u otra, han intentado o intentan resolver el mismo o similares objetivos, extrayendo de ellos las características más interesantes que podrían integrarse en nuestra solución.

1.1.1 Interoperabilidad

El primer problema planteado en esta tesis es el problema de la interoperabilidad de objetos, descrito en el Capítulo I, que supone la dificultad para que varios objetos se conozcan, interaccionen, se comuniquen y cooperen, mediante mecanismos de uso común, como puede ser la invocación de métodos. Este problema se vuelve tanto más grave cuantas más capas se presentan en el sistema, cada una de las cuales suele ofrecer un modelo de objetos distinto.

Este problema, presente en la mayor parte de los sistemas que incorporan TOO, supone una importante merma en el rendimiento al ser necesario cambios de paradigma que se implementan como capas adicionales.

La solución supone construir un sistema de soporte de objetos básico a todas las demás capas del Sistema OO. Sin embargo, la construcción de este sistema de soporte ha dado lugar a varias soluciones, no todas válidas. La solución a este primer problema se planteará en el Capítulo V y VI.

Mientras tanto, en el Capítulo II se exploran algunas de las tecnologías vistas para la construcción de Sistemas Operativos, soporte básico por excelencia en los sistemas actuales, sean estos OO o no.

1.1.2 Integración de la Computación en el entorno OO

El segundo problema que se plantea es la forma de integrar en el sistema de soporte de objetos, la computación de objetos.

En el Capítulo III se presentan los problemas de las abstracciones actuales ofrecidas por los Sistemas Operativos en uso. Este es un problema difícil de resolver, más aún cuando se verá que existen una serie de requisitos incontestables para la construcción de un entorno de computación OO adecuado.

1.1.3 Sistemas Relevantes

Por último, en el Capítulo IV se hace un breve repaso de algunos de los sistemas actuales más interesantes. No todos ellos son OO, sin embargo, de todos ellos se pueden extraer características interesantes para la construcción del sistema OO solución.

1.2 Solución

Se trata de la parte más extensa de todas y supone un largo camino hacia el planteamiento de la solución al problema planteado en la sección anterior.

En esta sección se determina la construcción de un SIOO que introduzca los conceptos de computación OO concurrente en las capas más básicas de soporte, ocultando a las capas superiores del sistema, la existencia de cualquier otra abstracción que no sea el objeto.

Para la construcción de este SIOO, se propone como aporte a los sistemas ya vistos una combinación de técnicas que, aunque están siendo exploradas por otros investigadores, no se combinan de la misma forma que se propone. Al final se estudia lo prometedora de esta combinación de técnicas, como son la OO, máquinas abstractas como núcleo básico de la computación y reflectividad como medio para dotar de flexibilidad al sistema.

1.2.1 SIOO: Máquina Abstracta Orientada a Objetos + Sistema Operativo Orientado a Objetos

En el Capítulo V se presenta la construcción de un SIOO que dé soporte básico a objetos como la solución a los problemas de interoperabilidad presentados en la sección anterior.

El SIOO se construye en torno a dos elementos básicos, una máquina Abstracta OO y un Sistema Operativo OO, que, juntos, ofrecen el soporte básico a la existencia, la ejecución y la comunicación de objetos.

La MAOO define un modelo de objetos básico heredado por todas las capas superiores que impone ciertas características a los objetos lo que facilita que estos puedan operar unos con otros al estar asentados sobre una base común.

El SOOO extiende esas características con otras como la distribución o la persistencia, dotando al modelo de propiedades no ya básicas, sino avanzadas. En el Capítulo VI se extiende la discusión acerca del soporte a objetos por parte del Sistema Integral, más concretamente, del SO.

La construcción de MAOO y SOOO se lleva a cabo aplicando varias de las técnicas vistas en el Capítulo II como son la OO, la construcción de un núcleo duro, que estaría representado por la máquina abstracta y la integración de este núcleo con el SOOO por medio de la Reflectividad.

1.2.2 Entorno de Computación Orientado a Objetos en el SIOO

Una vez determinada a grandes rasgos la forma que tendrá el SIOO, en los Capítulos VII, VIII y IX se afronta el otro gran problema que es la integración en este SIOO de la computación de objetos.

El problema general se planteó en el Capítulo III desechando todas las soluciones tradicionales al ser incompatibles con los requisitos del modelo de objetos planteados en el Capítulo V.

Además, la construcción de este entorno de computación está sujeto a ciertas imposiciones. En el Capítulo VIII se presentan los requerimientos funcionales, como la concurrencia entre y dentro de los objetos, la interacción entre objetos y otras.

En el Capítulo IX se discuten los requerimientos no funcionales, entre los que destaca la flexibilidad, en contraposición a muchos de los sistemas actuales.

1.2.3 Modelo de Objetos para la Computación

Como continuación de los Capítulos VII, VIII y IX, el Capítulo X plantea el modelo de objetos para la computación, basado en la integración de la abstracción de ejecución con la abstracción de objeto.

Esta integración se lleva a cabo en la capa más básica del SIOO, como es la máquina abstracta, que, de esta forma, ofrecerá una abstracción de objeto activo al resto del sistema, incluido el SOOO.

Con ello se logra que la única abstracción existente en el sistema sea el objeto, haciendo innecesaria la adición de otras como los procesos, etc., que no suponen más que una merma en el rendimiento al tener que trabajar con varias abstracciones de semántica bien distinta. Más grave que eso, supone la violación del principio de abstracción única y homogeneidad.

1.2.4 Integración de SOOO y MAOO: Reflectividad

En el capítulo XI se plantea la forma de integrar SOOO y MAOO. Esta integración debe ser, necesariamente, flexible, de forma que el SOOO extienda la máquina abstracta

por medio de objetos homogéneos con el resto de objetos del sistema. Esto da pie a que las aplicaciones puedan sustituir unos objetos por otros que ofrezcan la misma interfaz implementada de forma más adecuada a sus requerimientos.

Se verá que, si bien se aplica la técnica del micro núcleo o el exo-núcleo para la construcción del SIOO, donde todo gira en torno a un núcleo básico como la máquina abstracta, y se aplica la técnica de OO para el diseño y construcción tanto de la MA como del SO, será la técnica de la reflectividad, la que se ocupe de determinar la forma en que MAOO y SOOO interaccionan.

En el Capítulo XII se ofrece un estudio detallado de esta técnica y en el Capítulo XIV se aplica la misma a la integración del SOOO y la MAOO, definiendo una arquitectura reflectiva concreta para el SIOO.

1.2.5 MAOO

Se presenta a fondo la MAOO, cosa de la que se ocupa el Capítulo XV. Se describe cuál es la funcionalidad básica que ofrece, cómo es la abstracción de objeto que ofrece, cómo soporta esta abstracción, qué objetos internos define para soportarla, qué mecanismos ofrece para comunicarlos y qué soporte ofrece al sistema operativo.

Por otro lado, también se describe los elementos de la máquina que refleja al exterior (reflectividad estructural) y que son fundamentales, junto con la modificación de la comunicación entre objetos, para lograr la Reflectividad del Comportamiento gracias a la cual se permite que el Sistema Operativo influya y modifique el comportamiento de la misma.

1.2.6 SOOO

Construido siguiendo técnicas de OO, como se vio en el Capítulo VI, se describe en el capítulo XVI el conjunto de objetos que componen el Sistema Operativo y extienden la máquina abstracta.

Esta descripción comprende las acciones que llevan a cabo y la forma de comunicarse con otros objetos del Sistema Operativo y de usuario, así cómo, la utilización de los mecanismos de la máquina abstracta por parte de los objetos del sistema operativo, para definir una parte del comportamiento de la máquina abstracta.

1.3 Prototipo

En el Capítulo XVII se describe un prototipo, de los varios que se han implementado para el SIOO, construido siguiendo los principios de diseño básicos señalados en esta tesis.

En este prototipo no se hace hincapié en la eficiencia sino en la demostración de que la construcción de un sistema como el descrito es posible y que, efectivamente, soluciona los problemas planteados.

1.4 Conclusiones

Por último, en la sección de conclusiones de ofrece un resumen de los requerimientos y técnicas planteadas para la solución del problema de integración de la

computación en un entorno OO. Así como de la arquitectura reflectiva utilizada para combinar máquina abstracta y Sistema Operativos.

Capítulo I

Los Sistemas Operativos y el Paradigma de la Orientación a Objetos

En los últimos años se han llevado a cabo numerosos trabajos que estudian la integración de los Sistemas Operativos con una de las tecnologías que más desarrollo ha tenido recientemente, como es la Tecnología de Orientación a Objeto[CC91, CJM+89, KL93, Kra93, Sch93, SGH+89, Rus91a, Rus91b].

Sin embargo, esta integración no ha seguido un camino sencillo, sino más bien, un camino largo y tortuoso, plagado de soluciones intermedias y que todavía no ha llegado a su fin.

En este primer capítulo se establece el ámbito conceptual de esta tesis, centrándose en el paradigma de Orientación a Objetos como técnica válida para el diseño y desarrollo de Sistemas Operativos y rechazando las soluciones intermedias propuestas para dar soporte a las tecnologías OO sobre SO tradicionales.

A continuación, se mostrará la validez de la misma, justificando cómo un Sistema Operativo Orientado a Objetos es la solución idónea para la integración total y homogénea de las TOO en un sistema informático.

Para finalizar, se buscará el modelo de objetos más adecuado, que defina la comunicación entre objetos, el control de las actividades, en definitiva, el modelo de ejecución del sistema.

Para ello, se introducirá brevemente el problema de diseñar un entorno de computación como parte fundamental del Sistema Operativo que encaje, sin fisuras, con el paradigma de Orientación a Objetos.

1 El Paradigma de Orientación a Objetos como tecnología de desarrollo

En los últimos años se ha dedicado gran cantidad de trabajo al estudio e implantación de las tecnologías orientadas a objetos en los distintos ámbitos de la informática[CC91, Tho89, Weg90].

En estas obras se proponen posibles modelos de objetos adoptables que definen las características a cumplir por un entorno que quiera ser considerado Orientado a Objetos, sea este un lenguaje de programación, un Sistema de Gestión de Bases de Datos o un Sistema Operativo.

1.1 Modelo de Objetos Estándar

Sin embargo, es en [Boo91] donde se ha documentado el modelo de objetos más ampliamente difundido y aceptado como el que determina cuáles son las características que debe soportar un objeto estándar.

1.1.1 Encapsulación

Las técnicas de encapsulación de datos[AR84] sitúan juntos los datos con las funciones que operan en ellos restringiendo el acceso a tales a datos exclusivamente a través de tales funciones.

En el paradigma de Orientación a Objeto, la unidad de encapsulación de datos es el objeto. Cada objeto encapsula un conjunto de variables, **estado**, y un conjunto de métodos utilizados para acceder y alterar las variables, **interfaz**. La única forma de llevar a cabo alguna función sobre los datos es mediante la invocación de alguno de los métodos que componen la interfaz del objeto. De esta forma se preserva el estado del objeto entre invocaciones a métodos.

Las técnicas de encapsulación de datos se utilizan para incrementar la modularidad, la mantenibilidad y la fiabilidad de los programas.

1.1.2 Herencia

En el paradigma de la orientación a objetos, aparece de forma natural el concepto de **clase** para expresar las características comunes entre objetos con idéntico comportamiento que se diferencian únicamente por el valor de sus variables. Cada clase define la interfaz y encapsula la implantación de las instancias de esa clase.

La **herencia**[BDM+73] es una técnica que permite a un conjunto de clases compartir partes de una interfaz común e incluso de una implantación común (métodos y variables).

La utilización de la herencia permite separar la interfaz de la clase de su implantación, hecho que se logra utilizando clases abstractas/concretas y que tendrá una enorme repercusión a la hora de conseguir características muy deseables en un Sistema Operativo como la **portabilidad**.

1.1.3 Polimorfismo

El polimorfismo permite que diferentes objetos respondan al mismo mensaje y será el sistema, en tiempo de ejecución, el que decida la implantación adecuada del mensaje en función de la instancia concreta del objeto en tiempo de ejecución.

Permite escribir diferentes implantaciones para una misma interfaz, y la decisión de qué implantación utilizar, podría tomarse en función de los parámetros recibidos durante la invocación. Es posible que un método reciba diferentes argumentos y se comporte de forma diferente según los argumentos recibidos.

El polimorfismo a través de la herencia consiste en la redefinición de un método de tal forma que, cuando se invoca un método en un objeto, determinar qué método se ejecutará como respuesta a dicho mensajes, es una decisión que se retrasa hasta el momento de la ejecución.

El polimorfismo proporciona un alto grado de **flexibilidad** a la hora de diseñar y reconfigurar software OO, permitiendo la adición de nuevos componentes y el reemplazamiento de otros ya existententes por medio de nuevas clases que implementan la interfaz antigua.

1.2 Razones para migrar a un entorno OO

Además de documentar las distintas características que un entorno debe cumplir para ser OO, también se ha realizado un gran esfuerzo en documentar las razones que han llevado a la migración desde un entorno orientado a procedimiento a un entorno OO, si bien es en [Boo91] donde estos móviles se presentan de forma más extensa.

Son muchas las ventajas derivadas de desarrollar aplicando las TOO. Se hace aquí un primer esbozo de las más conocidas aunque en el Capítulo II se hará un estudio más detallado de las ventajas que ofrecen las TOO al desarrollo concreto de un Sistema Operativo.

Reutilización del código

Las técnicas de OO consiguen un alto grado de reutilización a través de la encapsulación y la herencia.

Modularidad

Proporcionan una clara separación entre interfaz e implementación lo que contribuye a que los sistemas sean fáciles de entender y modificar.

Escalabilidad

Facilidad de evolución

Dada la arquitectura de las aplicaciones OO, basada en la modularidad del código alrededor de trozos básicos (objetos), es posible reemplazar alguno de ellos por otro que, si bien soporta la misma interfaz, puede llevar a cabo una implementación distinta de la misma.

Flexibilidad

Gracias a técnicas como el polimorfismo es posible conseguir un alto grado de flexibilidad a la hora de diseñar y reconfigurar sistemas operativos orientados a objeto, además de ser vital para el diseño de código reusable. El polimorfismo permite la adición de nuevos componentes y permite reemplazar componentes ya existentes por otros.

Todas estas razones, aún siendo importantes a la hora de diseñar cualquier tipo de aplicación, cobran gran trascendencia a la hora de diseñar y desarrollar sistemas software tan grandes y complejos como los sistemas operativos.

2 Sistemas Operativos Orientados a Objeto: Una Plataforma Necesaria

Hoy por hoy, la incorporación del paradigma de la orientación a objetos no se está llevando a cabo de forma integral en todos los componentes de un sistema informático,

sino que se aplica de manera **parcial** sólo en alguno de ellos. De hecho, como se verá posteriormente, son los Sistemas Operativos uno de los campos en el que la aplicación de las tecnologías OO ha tenido una incidencia más tardía, siendo así que los trabajos sobre Sistemas Operativos OO, están datados todos en fechas recientes.

Sin embargo, otros campos como los lenguajes de programación, han sido muy receptivos a esta nueva tecnología provocando que, lenguajes OO lleven ejecutándose desde hace años sobre sistemas Operativos tradicionales orientados a procedimiento, produciéndose una serie de problemas que se estudiarán a continuación.

Estos sistemas Operativos orientados a procedimiento no proporcionan a los lenguajes OO el soporte adecuado, produciéndose carencias que, continuamente, se solucionan con parches que merman el rendimiento global del sistema.

Además de esta **integración parcial** de las tecnologías orientadas a objetos, es necesario destacar la **falta de uniformidad** con que esta se está llevando a cabo. [Sch93]. Esto provoca que, elementos que adoptan el paradigma de OO lo hagan de tal forma que resulte imposible, o al menos muy difícil, que puedan comunicarse.

La construcción de un Sistema Operativo OO viene a solucionar también este problema ofreciendo una plataforma básica de soporte uniforme para todos los demás elementos del sistema que se ejecuten por encima.

A lo largo de este apartado se profundizará en estos aspectos, señalando las causas y consecuencias del actual **caos de objetos**.

2.1 Integración no Completa del paradigma de la OO

El paradigma de la OO está encontrando algunos escollos a la hora de incorporarse a todas las capas de un sistema. Estos problemas derivan, fundamentalmente, de dos causas, que se pueden resumir en la **integración no completa del paradigma**.

2.1.1 Adopción Parcial

Existen campos en los que el paradigma de Orientación a Objetos está plenamente adoptado y ampliamente difundido. Así, la integración en el campo de los lenguajes de programación está muy estudiada y en un estado muy avanzado existiendo multitud de lenguajes de programación que han adoptado el paradigma de la orientación a objetos (C++[Str93], Java[GJS97, AG97], etc.).

También es muy común la utilización de sistemas de gestión de bases de datos, herramientas CASE, interfaces gráficas de usuario convencionales en las que funcionan aplicaciones cuya interfaz es orientado a objetos, etc. Y, de la misma forma, existen metodologías de análisis y diseño orientadas a objeto.

Sin embargo, los sistemas operativos más ampliamente difundidos ofrecen a sus aplicaciones una interfaz universal y no Orientada a Objetos para utilizar los servicios del sistema. Este problema ya ha sido señalado por [Sch93] y varios trabajos acerca de la adopción del paradigma al campo de los Sistemas Operativos intentan paliar esta carencia.

Recientemente, la cantidad de trabajos relacionados con la integración de las TOO y los SO ha aumentado considerablemente y así, es posible citar los trabajos derivados del diseño y construcción de Choices[CJM+89, CIM+93], Apertos[Yok92] y su precursor

MUSE[YTT89], Sos[SGH+89], Guide[BBD+91], Tigger[Cah96b] y otros que se irán referenciando a lo largo de esta tesis.

2.1.2 Adopción No Uniforme

Incluso en el caso de campos en los que el paradigma de Orientación a Objetos está plenamente adaptado, como pueden ser los lenguajes de programación, la adopción se llevó a cabo de manera no uniforme, de forma que cada lenguaje adopta un modelo de objetos particular, con características propias, en lugar de un modelo estándar con características básicas comunes a todos.

2.2 Problemas de la adopción No completa del paradigma OO

La falta de uniformidad en la adopción del paradigma OO, añadido a la carencia de un soporte adecuado por parte del Sistema Operativo, se va a traducir en capas intermedias de software que adapten el entorno de desarrollo y de ejecución OO a un sistema operativo estructurado en base a abstracciones tradicionales[Kon93].

2.2.1 Problema de la desadaptación de impedancias o salto semántico

Las aplicaciones OO trabajan, normalmente, sobre SO tradicionales no OO, que presentan una serie de carencias para su soporte.

Abstracciones no adecuadas

Los sistemas operativos tradicionales no ofrecen la abstracción de objeto. En su lugar, presentan abstracciones como los procesos o los ficheros. Así, las capas OO que trabajan por encima, se ven obligadas a construir sus abstracciones utilizando para ello las que ofrece el sistema operativo.

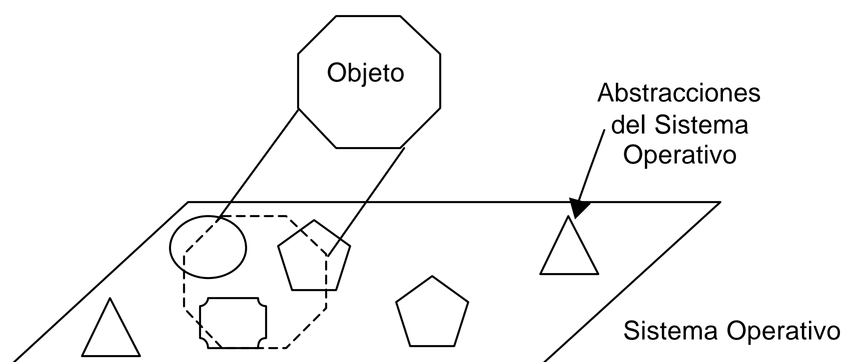


Figura 1.1 Abstracciones Poco Adecuadas para el Soporte de Objetos

Interfaz no OO del sistema operativo

La interfaz que los Sistemas Operativos tradicionales ofrecen a los objetos es mediante las llamadas al sistema y no la invocación a métodos. De esta forma, el acceso a los mecanismos del sistema operativo se hace de una forma distinta a los servicios ofrecidos por objetos haciéndose evidente la diferencia entre el SO y la aplicación.

Esta **falta de transparencia** no tendría lugar si las aplicaciones accediesen a los servicios del Sistema Operativo y a los servicios ofrecidos por otros objetos de la aplicación de la misma forma.

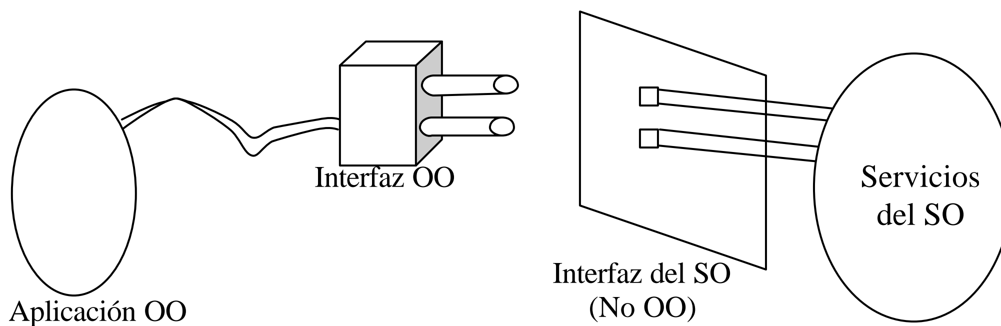


Figura 1.2 Interfaz Inadecuada

Dificultad de Comunicación de alto nivel entre elementos situados en espacios de direcciones diferentes

Para el desarrollo de aplicaciones en entornos distribuidos con el paradigma OO, la arquitectura cliente-servidor se lleva a cabo con objetos cliente que solicitan servicios proporcionados a través de los métodos de objetos servidores. El mecanismo de comunicación debe ser un mecanismo de alto nivel que no rompa el paradigma de la orientación a objeto.

El problema de la comunicación entre objetos es debido a que el uso de éstos suele estar limitado al programa al que pertenecen y, por tanto, pertenecen a distintos espacios de direcciones. Hay que recurrir, entonces, a mecanismos de bajo nivel, como los pipes, soportados por el sistema operativo.

2.2.2 La interoperabilidad entre distintos modelos de objetos

La **interoperabilidad** es la habilidad de dos o más entidades, como programas, objetos, aplicaciones, etc., de comunicarse y cooperar independientemente de las diferencias de lenguaje de implementación, entorno de ejecución o abstracciones del modelo. Esto supone no sólo el intercambio de información, sino el uso de recursos disponibles en otros entornos[Kon93].

Es posible, incluso habitual, que distintos entornos implementen distintos modelos de objetos. Esto supone que cada compilador implementa internamente un modelo de objetos distinto, lo que produce una incompatibilidad entre los binarios de dos aplicaciones escritas en distintos lenguajes o incluso, en un mismo lenguaje pero con compiladores de distintos fabricantes.

Esto implica que no siempre es posible asumir que la funcionalidad solicitada por el cliente encaja exactamente con la funcionalidad ofrecida por el servidor.

En este caso, las distintas capas no serán capaces de comunicarse a pesar de estar basadas en el paradigma de la OO.

2.3 Soluciones Parciales a los problemas de interoperabilidad

Actualmente, la solución a este problema se lleva a cabo introduciendo capas intermedias entre los elementos que utilizan la orientación a objetos y el sistema operativo que no lo utiliza[Byt95].

Una de las soluciones más populares es añadir al sistema un **gestor de objetos** que permita que un objeto pueda invocar de manera transparente un método de otro objeto situado en un espacio de direcciones diferente y de manera independiente tanto del lenguaje como del sistema operativo.

Figura 1.3 Enlace No Directo entre Aplicaciones y Sistema Operativo

Para el desarrollo de un gestor de objetos es necesario establecer una serie de convenios globales acerca de la semántica de los objetos servidores: manera de expresar la funcionalidad, forma de realizar la comunicación (síncrona, asíncrona), etc., además de establecer una correspondencia entre esta semántica de los objetos servidores y la implantación que se lleve a cabo de los mismos.

2.3.1 Corba

El ejemplo más relevante es CORBA[OMG96, OHE97], capa común a través de la cual, los objetos pueden intercambiarse mensajes y recibir respuestas de forma transparente, permitiendo así la interoperabilidad entre aplicaciones ejecutándose en distintas máquinas en un entorno distribuido.

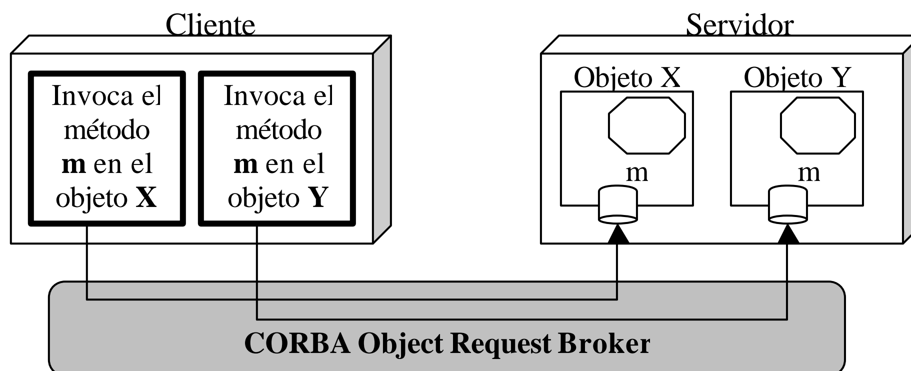


Figura 1.4 CORBA

Corba salva el salto semántico entre los objetos clientes y los servidores mediante el uso de un IDL que obliga a los clientes a utilizar una interfaz predefinida, lo que,

claramente, limita la libertad del cliente de acceder a servidores alternativos que proporcionen el mismo servicio con una interfaz diferente.

2.3.2 SOM

SOM (*System Object Model*) [IBM96a, IBM96b, ~SOM, TD99] es una Tecnología Orientada a Objetos nacida en el seno de IBM con el objetivo de establecer un mecanismo que permita la **interoperabilidad** entre objetos independientemente del lenguaje de programación o el sistema operativo en que se hayan definido.

Así pues SOM es un estándar binario para objetos, concebido para resolver el problema del fuerte acoplamiento binario entre una aplicación y las bibliotecas de clases que ésta utiliza.

Separación Interfaz e Implantación

Para lograr la interoperabilidad, SOM establece una estricta separación entre la definición de la interfaz de una clase y su implantación, es decir, entre los métodos que soporta esa clase y los procedimientos que implementan estos métodos.

Mientras la interfaz del objeto es conocida por los clientes, el conjunto de procedimientos que se ejecutarán como respuesta a la invocación de los métodos de la misma, es completamente transparente a los mismos, que sólo conocerán los servicios ofrecidos por el objeto servidor.

SOM define un IDL (*Interface Definition Language*) en el que el programador describe la interfaz de la clase. Definida la interfaz, el programador puede detallar la implantación de las clases en un lenguaje de programación cualquiera.

Reutilización

Gracias a la separación interfaz/implantación, los clientes únicamente necesitan conocer la interfaz de la clase, siendo así totalmente inmunes a cambios en detalles de implantación de los procedimientos que implementan los métodos de la clase que utilizan, no necesitando recompilación ni recarga ante estos cambios.

Modelo de Objetos

El modelo de objetos de SOM proporciona encapsulación, herencia múltiple con mecanismos para resolver ambigüedades, polimorfismo, metaclases, diferentes modos de resolución de métodos y creación dinámica de clases. Además SOM proporciona una jerarquía básica de clases a partir de las cuales se generan el resto de clases y metaclases.

2.3.3 DSOM. Distributed SOM

DSOM[IBM96a, IBM96b, ~SOM, Izq99] es una extensión de SOM que nace con el objetivo de añadir a la independencia de lenguaje y Sistema Operativo proporcionada por SOM, independencia de localización de los objetos, permitiendo la operación y comunicación entre objetos situados en diferentes espacios de direcciones o incluso en diferentes máquinas. Con esta evolución, ambas han quedado íntimamente ligadas, sin que sea posible ya diferenciar cuando se habla de SOM a qué se está refiriendo.

Transparencia de Acceso y Localización

DSOM permite la interacción entre objetos situados en diferentes espacios de direcciones que pueden residir en la misma o diferente máquina. La localización del objeto no importa al objeto cliente, siendo así que el cliente puede acceder de la misma forma independientemente de que su localización sea local o remota. Es más, el cliente no sabe si el objeto es local o remoto e incluso el objeto puede cambiar su localización de una ejecución a otra sin que la aplicación necesite ni modificaciones en su código fuente ni recompilación.

Para lograr la interoperabilidad, DSOM se construye como un ORB que se ajusta a la especificación de la Common Object Request Broker Architecture (CORBA) publicada por el Object Management Group (OMG) y X/Open. También se ajusta a la especificación de CORBA Internet Inter-ORB Protocol (CORBA IIOP), que permite la interacción con otros ORB que se ajusten a CORBA IIOP.

2.3.4 COM

COM (*Component Object Model*)[Box98, AA99] de Microsoft, es una especificación de cómo crear componentes y cómo construir aplicaciones que los usen.

Un **componente** se puede definir como una unidad funcional que sigue el principio de encapsulación definido por el paradigma de OO. Para proporcionar su funcionalidad, los componentes exponen conjuntos de funciones que pueden ser invocados por clientes potenciales y que conforman el único modo en que un cliente puede utilizar el componente.

Interoperabilidad

Como cualquier otra especificación, COM define una serie de estándares que permiten la interoperabilidad de componentes.

- Un estándar binario para invocar las funciones de los componentes.
- Un componente puede tener varias interfaces, que agrupan funciones, y una interfaz base, denominada **IUnknown** que proporciona:
  - Un mecanismo para que los componentes puedan descubrir, dinámicamente, las interfaces soportadas por otros componentes (**QueryInterface**).
  - Control sobre el ciclo de vida de los componentes, basado en la cuenta de referencias (**AddRef** y **Release**).
- Un mecanismo para identificar, unívocamente, clases e interfaces (**GUID**).

Como **estándar binario**, no depende de ningún lenguaje de programación.

Reusabilidad

Uno de los principales objetivos de un modelo de objetos es capacitar a los diseñadores de componentes y a los programadores para reutilizar y extender componentes proporcionados por otros desarrolladores, como parte de sus propios componentes.

Para conseguir la reusabilidad, COM soporta dos mecanismos:

- Contención/Delegación. El componente externo se comporta como un cliente del interno, para lo que usa sus servicios para su propia implantación, delegando parte

de su implantación. Es sencillo de implantar, simplemente un componente contiene a otro.

- Agregación. Puede entenderse como una especialización de la contención. El componente externo expone las interfaces del interno, como si las implantase él.

2.3.5 Ventajas de la solución

La principal ventaja que proporcionan estas capas es **permitir la comunicación entre objetos construidos basándose en distintos modelos de objetos**, con distintos lenguajes y a través de un sistema operativo no orientado a objetos. Estas capas simplemente traducen de un paradigma a otro, posibilitando el entendimiento y la interoperabilidad de los elementos.

2.3.6 Inconvenientes

Esta solución al problema de la interoperabilidad, no deja de ser un parche sobre un sistema operativo que es incapaz de comunicar dos modelos diferentes. La adición de parches intermedios provoca que esta no sea una solución global lo que se traduce en una serie de inconvenientes que se citan a continuación.

Disminución del rendimiento

Al ser necesario atravesar todas estas capas de software, se provocan una sobrecarga considerable al tener que añadir llamadas adicionales para salvar el salto entre distintos modelos de objetos, lo que produce una disminución del rendimiento global del sistema.

Falta de transparencia de las soluciones

Estas soluciones, en la práctica, no son todo lo transparentes que deberían ser de cara al usuario. Por ejemplo, la escritura de objetos que vayan a funcionar como servidores suele realizarse de manera diferentes del resto de los objetos.

Falta de Uniformidad Conceptual en torno a la OO

Dado que el sistema operativo subyacente sigue siendo tradicional, se hace necesario usar dos paradigmas de programación diferentes: procedimental y de orientación a objetos.

- Por un lado, el programador normalmente trabaja utilizando el paradigma de la orientación a objetos, pero usará el paradigma orientado a procedimiento cuando necesite utilizar los recursos del sistema operativo (vía llamadas al sistema), lo cual provoca una pérdida de productividad.
- Por otro lado, para la implantación de estas capas que ofrecen soporte de objetos, se necesitan también los servicios del sistema operativo, con lo que nos encontramos con un salto entre paradigmas. Dicho salto provocará de nuevo una pérdida de rendimiento.

Además, aunque la capa de soporte sea OO, el concepto de objeto difiere en el lenguaje y la capa de soporte.

2.4 SO + OO: Una buena pareja

Todas las soluciones anteriores no son más que parches que tratan de dar una solución, al menos en parte, al problema planteado. Sin embargo, aún solucionándolo, traen consigo problemas adicionales intrínsecamente ligados a la adición de capas software y, por tanto, de muy difícil solución.

La aproximación más razonable pasa por proporcionar el soporte de objetos como parte del sistema operativo, construyendo lo que se ha dado en llamar un **Sistema Operativo Orientado a Objeto** (**SOOO**), tal y como se propone en otros trabajos como[LJP93]. Esta es, sin duda, la visión más prometedora para dar soporte a las nuevas tecnologías Orientadas a Objeto.

Sistemas Operativos con Soporte para Objetos

Un Sistema Operativo completamente Orientado a Objeto se puede definir como aquel que soporta el concepto de objeto, incluidos los objetos definidos a nivel usuario, como entidades con una semántica fuerte y bien definida[Sha91b].

Un SOOO se compone de un conjunto de objetos identificables y capaces de comunicarse que ofrecen la funcionalidad típica de un Sistema Operativo a través de la interfaz.

La principal función de un SOOO es ofrecer un soporte homogéneo a los objetos, como identificación de objetos, almacenamiento de objetos, migración a la vez que permitir el acceso y la comunicación de los objetos mediante un mecanismo uniforme, sean estos objetos definidos por las aplicaciones u objetos del sistema[Cah96b].

Un SOOO proporciona a las aplicaciones que se ejecutan sobre él un entorno orientado a objetos con lo que se consigue la ventaja adicional de soportar a un nivel básico un modelo de objetos homogéneo heredado por todos los niveles superiores.

Sistema Operativo que utiliza el Modelo del Lenguaje

Los primeros Sistemas Operativos Orientados a Objeto sufrían sin embargo, de una excesiva dependencia del modelo del lenguaje. Más aún, lo más frecuente era que el SO se estructurase de modo OO escribiéndose en algún LOO (por ejemplo, en C++). Sin embargo, esta estructura se perdía durante la construcción del sistema de tal forma que las aplicaciones sólo podían usar la interfaz tradicional de los SO y perdían los beneficios de la estructura OO. El ejemplo perfecto de tal situación es Choices[MCK91].

En el capítulo II se estudian las posibilidades de desarrollo de SOOO y la forma en que estos pueden dar soporte a los objetos. Más tarde, en el capítulo IV se hace una revisión de los SOOO más destacables.

3 SO OO: Primer Objetivo

Diseño y Construcción de una plataforma de soporte de objetos

El primer objetivo de esta tesis es encontrar un medio adecuado para la organización de un nuevo Sistema Operativo que dé soporte a los objetos definidos por las aplicaciones, a la vez que ofrece un mecanismo uniforme de acceso y comunicación con los objetos de las aplicaciones y de solicitud de servicios al propio sistema operativo.

Se mostrará que la estructuración del sistema operativo como conjunto de objetos es, sin duda la forma más adecuada para obtener una visión homogénea del sistema global.

Modelo de Objetos

A la hora de definir la plataforma de soporte, será necesario determinar el conjunto de características de que dotará a los objetos. Al colocarse estas en una capa básica de soporte del sistema, serán heredadas por los objetos de las capas superiores, facilitando la interoperabilidad.

Adaptación a los requerimientos de las aplicaciones

Los sistemas software modernos se enfrentan continuamente con una gran variedad de nuevos requerimientos de forma que demandan del sistema operativo nuevos servicios.

Los sistemas operativos, tradicionalmente, se estructuran en espacio de usuario y espacio del núcleo. El núcleo proporciona algunos servicios transparentemente a las aplicaciones, por ejemplo, espacios de direcciones o memoria virtual.

Sin embargo, cambiar un servicio ya existente en el sistema operativo o adecuar éste de algún modo a las aplicaciones en ejecución no es una tarea sencilla, ni siquiera en los micronúcleos que enarbolaron la bandera de la adaptación sencilla.

Reflectividad como una técnica prometedora

Uno de los mecanismos más prometedores para la estructuración de un sistema orientado a objetos que facilita la adaptación del mismo en tiempo de ejecución es la reflectividad[Golm97, GKS94], mecanismo que encaja a la perfección con el paradigma de la OO y que se tratará profundamente en los Capítulos II y, fundamentalmente el XII.

Ese nuevo paradigma de estructuración, que propugna la separación del código funcional (nivel base) del no funcional (meta nivel), es fundamental a la hora de proporcionar adaptabilidad al sistema.

La separación de código, hace posible reemplazar el código no funcional adaptando de esta forma el sistema a nuevos requerimientos de forma dinámica sin implicar un cambio en el código base.

3.1 Ventajas de la construcción de un SOOO para solucionar los problemas de desadaptación e interoperabilidad

Al darse soporte a los objetos desde el sistema operativo, se solucionan de forma más adecuada los problemas originales. En el Capítulo VI se dará una justificación detallada.

Ahorro de capas intermedias

Se evita la proliferación de capas intermedias y cambios de paradigma, que disminuían el rendimiento.

Modelo de objetos estándar y aceptado por todas las capas

En primer lugar, todos los compiladores generarían objetos en el modelo estandarizado que soporta el sistema operativo, con lo que podrían usarse directamente desde cualquier otro lenguaje y compilador.

Transparencia de versiones

En segundo lugar, dado que el uso de todos los objetos es gestionado por el sistema operativo, el cambio en la implantación de un objeto no afectará a los clientes que lo usen. Será responsabilidad del sistema operativo el que los clientes utilicen de manera transparente la nueva implantación.

Comunicación uniforme de alto nivel entre objetos

Por último, el sistema operativo proporcionará mecanismos nativos de comunicación de alto nivel entre objetos.

4 Computación en sistemas Orientados a Objetos

El segundo objetivo de esta tesis es definir el modelo de objetos que regirá la computación en el sistema orientado a objetos anterior. En los capítulos VII, VIII y IX se definen los objetivos que debería cumplir un sistema de computación OO y en el Capítulo X se establece una comparación entre los dos modelos más adecuados el **modelo activo** y el **modelo pasivo** estableciendo las ventajas e inconvenientes de cada uno.

Tradicionalmente, existen modelos de computación como el **modelo de proceso** o el modelo de tarea/hilo (del inglés, *task/thread*). Estos modelos construyen entidades independientes (procesos, tasks, threads) con unos mecanismos de interacción predefinidos que representan al nivel del sistema operativo las distintas actuaciones de la máquina hardware subyacente.

Falta de abstracciones adecuadas en los sistemas operativos tradicionales.

Las abstracciones ofrecidas por los Sistemas Operativos tradicionales se caracterizan principalmente por ser inadecuadas en un entorno OO (Ver Apartado 2.2.1).

Los procesos son espacios de direcciones protegidos únicamente accesibles a través del código definido en ellos. En la OO, ese papel lo juegan, en gran medida los objetos.

El salto semántico entre las abstracciones tradicionales y los objetos hacen que las primeras se hayan revelado poco adecuadas para dar soporte al paradigma de OO al necesitar mecanismos intermedios que nos permitan comunicarlas.

Inflexibilidad y falta de adaptación

Otro inconveniente de las abstracciones tradicionales es que están completamente integradas(del inglés, *hard-coded*) en el sistema operativo ofreciendo políticas de gestión de los recursos que causan el menor daño posible a la mayor parte de las aplicaciones. Lo que está lejos de ser una buena política[Cah96a]. Son sistemas inflexibles.

Uno de los objetivos de esta tesis es ofrecer un sistema de computación flexible, en el que el usuario pueda adaptar el entorno de ejecución entorno a las necesidades de su aplicación.

Nuevamente, como se verá en el capítulo XIV, la reflectividad será el mecanismo a utilizar para dotar al sistema operativo de la flexibilidad deseada.

Capítulo II

Tecnologías para la Construcción de un Sistema Operativo

El primer objetivo a cumplir por esta tesis es determinar la estructura interna más adecuada aplicable al nuevo sistema operativo con el fin de que este se convierta en una plataforma de soporte uniforme para los objetos definidos, a la vez que el propio sistema operativo se estructura de forma que sus servicios sean requeridos de igual forma que los de cualquier otro servidor ajeno al sistema.

A la hora de construir un Sistema Operativo son varias las posibles estructuras internas que se pueden aplicar al mismo. Se pueden citar cinco tecnologías diferentes, no excluyentes, cuya combinación permite, en muchos casos, conseguir los objetivos perseguidos de modularidad, adaptabilidad, flexibilidad, etc. [Cah96a].

Estas cinco tecnologías son:

- Tecnología microkernel
- Sistemas operativos específicos para la aplicación
- Orientación a objetos
- Familias de programas
- Reflectividad

En este capítulo, se realizará un estudio de las más destacables. En capítulos posteriores se verá cómo, la solución final, importa características de muchas de ellas, fundamentalmente de la Orientación a Objetos.

1 MicroNúcleos

El desarrollo reciente en la arquitectura de los sistemas operativos ha estado dominado por la transición entre los denominados sistemas operativos monolíticos y la tecnología micronúcleo, (del término inglés, *microkernel*)[Gie90].

Esta última tecnología emergió al final de la década pasada y se volvió muy popular cuando OSF (Open Software Foundation) eligió un micronúcleo, Mach[Ras86, ABG+86], para su OSF/1 y USL (UNIX Systems Laboratories) eligió el micronúcleo Chorus[RAA+88] para desarrollar sobre él la última versión de UNIX System V[Che84, Che88].

1.1 Organización del Sistema Operativo

Esta tendencia consiste en descomponer el sistema operativo en un **núcleo básico** o **micronúcleo** y un conjunto de **componentes**, en lugar de utilizar la tradicional estructura monolítica.

Estos componentes se estructuran como un conjunto modular de **servidores**, que se ejecutan como procesos independientes en modo usuario, realizando tareas reservadas tradicionalmente al sistema operativo y comunicándose vía algún tipo de paso de mensajes.

Por su parte, el **micronúcleo** proporciona a los servidores servicios básicos genéricos, independientes de cualquier sistema operativo particular y básicos y que sería complicado, peligroso o costoso proporcionar en un servidor.

1.2 Definición y Funciones

Un **micronúcleo** es pues el núcleo de un sistema operativo que se responsabiliza, fundamentalmente, de implementar el servicio de comunicación entre los procesos lo que permite a los servidores comunicarse unos con otros e intercambiar datos independientemente de si están en un multiprocesador o en una red de computadoras.

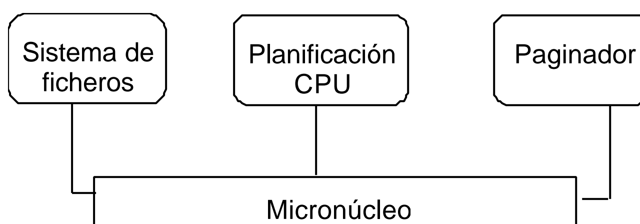


Figura 2.1 Estructura general de un Sistema Operativo con Micronúcleo

Además de la comunicación, el micronúcleo se encarga de proporcionar a los servidores soporte básico manipulando los recursos del sistema de más bajo nivel e implementando las abstracciones de bajo nivel como tarea (del inglés *task*), hilo o hebra (del inglés *thread*), espacio de direcciones virtuales, y puertos para la comunicación de procesos [Tan94, Var94, Cah96a, TNO92].

Las abstracciones de mayor nivel como una abstracción particular de proceso, así como los servicios del sistema como gestión de ficheros o redes, se implementan normalmente como servidores a nivel usuario.

Esta combinación de servicios elementales forma una base estándar que soporta todas las demás funciones específicas del sistema operativo. Estas funciones pueden configurarse en servidores (uno o varios) que gestionan los demás recursos físicos y lógicos de un sistema tales como ficheros, dispositivos y servicios de comunicación de alto nivel.

No obstante, existen bastantes controversias acerca de lo que se implementa en el núcleo y lo que no. De hecho, existen diferencias notables en el diseño de unos fabricantes a otros. De ahí que la cuestión de diseño en la arquitectura micronúcleo, sea la definición de los servicios que deben aparecer en le núcleo.

La interfaz del micronúcelo debe resultar de un compromiso entre el soporte total de estas funciones, claves para cualquier sistema operativo, y la posibilidad de

personalización, con el fin de adaptar el sistema a los requerimientos de áreas específicas de aplicación y sacar provecho de la arquitectura particular subyacente.

1.3 Ventajas e Inconvenientes de los micronúcleos

1.3.1 Ventajas

La introducción de la arquitectura microkernel ha venido motivada por la esperanza de una mayor portabilidad, una mejor adaptabilidad, flexibilidad, y modularidad a costa, normalmente de un peor rendimiento en ciertos casos.

De entre todas las ventajas que proporciona la estructuración de un sistema operativo como un micronúcleo y un conjunto de servidores a nivel usuario, se pueden destacar las siguientes [Cah96a, Tan94].

Adecuación de las características y eliminación de las no necesarias

Los micronúcleos promueven la separación entre las abstracciones de bajo nivel soportadas por el núcleo y las de alto nivel implementadas en servidores que viven a nivel usuario.

Esta separación permite que el sistema operativo pueda dar soporte a distintas personalidades simultáneamente lo que proporciona la flexibilidad necesaria para soportar las abstracciones adecuadas en cada caso evitando la deficiencia de características.

Esta separación también permite a los desarrolladores de aplicaciones elegir entre implementar directamente en la interfaz ofrecida por el microkernel y la ofrecida por alguna personalidad, evitando de esta forma la presencia de características innecesarias.

Incremento de la flexibilidad

Este incremento se promueve desde dos aspectos:

- **Soporte para el desarrollo de servidores**. En comparación con el diseño de los sistemas operativos monolíticos, los microkernels ofrecen una mayor flexibilidad permitiendo que partes del sistema operativo sean implementadas como servidores a nivel usuario [Cah96a].

  Un microkernel ofrece la posibilidad de poder modificar aquellos componentes del sistema operativo implementados como servidores lo que resulta mucho más sencillo que modificar cualquier componente o política integrado en un sistema operativo monolítico.

- **Colaboración de las aplicaciones en ciertos aspectos**. Los micronúcleos ofrecen también, normalmente, la posibilidad de que algunos aspectos de las abstracciones que ellos proporcionan sean implementados en colaboración con las aplicaciones. El ejemplo mejor conocido es el uso de paginadores externos para soportar el sistema de memoria virtual. De esta forma implementaciones nuevas o modificaciones de servicios particulares pueden desarrollarse fácilmente.

Modularidad

La comunicación entre los distintos procesos servidores (por ejemplo, sistema de fichero) y el micronúcleo se consigue utilizando el mecanismo de paso de mensajes lo cual enfatiza el diseño modular y la ocultación de la información.

Reutilización transparente de los componentes

La encapsulación proporcionada por el paso de mensajes significa que diferentes servidores pueden ser reutilizados bajo diferentes situaciones fácilmente. Con un planteamiento monolítico es muy complicado aislar el planificador de la CPU y moverlo a otra máquina.

En este aspecto, la combinación de micronúcleo y OO se ha revelado como una tecnología prometedora para el desarrollo de SO.

Portabilidad del hardware

Uno de los principales problemas para sacar una máquina nueva al mercado es el de proporcionar un sistema operativo para ella. Mediante el planteamiento de micronúcleo, en el que toda la información acerca del hardware está en el núcleo, esto sólo implica portar el micronúcleo, mientras que el resto de los servicios por encima son independientes de la plataforma.

Distribución transparente

Otra ventaja proporcionada por el uso de paso de mensajes es que el usuario no necesita conocer dónde está localizado el proceso servidor. Cuando un programa realiza una petición para abrir un fichero, el servidor que atiende esta petición podría estar localizado en una máquina diferente. Las facilidades de comunicación entre procesos proporcionadas por el micronúcleo gestionan el reparto de mensajes sin que el servidor o el usuario se den cuenta. Ver figura 2.2.

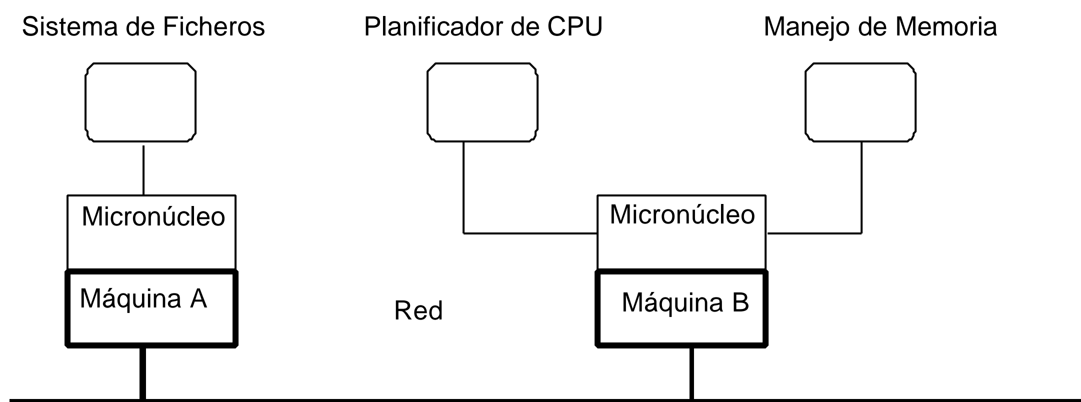


Figura 2.2 Micronúcleos y Distribución

Duplicación de Servicios y Tolerancia a Fallos

La posibilidad de distribuir los servicios de forma modular en forma de servidores permite la duplicación de servicios. Este hecho favorece dos características importantes en un sistema operativo.

- **Tolerancia a fallos**. Si falla un servidor, inmediatamente el otro se encarga de su trabajo evitando caídas del sistema, en ocasiones sumamente importante.
- **Equilibrio de carga**. En lugar de tener un servidor muy cargado es posible equilibrar la carga entre dos procesos, elevando la velocidad de respuesta.

1.3.2 Limitaciones de los micronúcleos

Adaptabilidad de grano grueso

Aunque un sistema micronúcleo permite la adaptabilidad del sistema operativo a las necesidades actuales, tal adaptabilidad se basa en el reemplazo de servidores y provoca que todos los clientes de ese servidor se vean afectados[KL96].

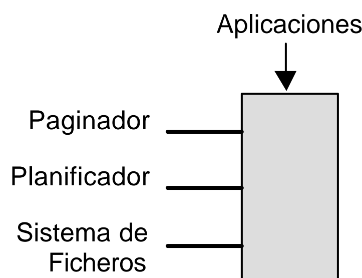


Figura 2.3 Adaptabilidad en ciertos aspectos

Bajo grado de incrementalidad

La incrementalidad se puede definir como el **grado de proporcionalidad entre el esfuerzo necesario para especializar una implantación respecto a la cantidad de cambios necesarios**.

Los sistemas operativos con arquitectura micronúcleo presentan un bajo grado de incrementalidad, fundamentalmente, por dos razones.

- En primer lugar, al ser los servidores la unidad de reemplazo, el grado de incrementalidad será menor que el deseado dado que no hay un soporte específico para modificar un servidor concreto para una aplicación.
- La segunda razón por la que se puede afirmar que la incrementalidad es pobre, es porque cada servidor que se construya, debe ser construido desde cero.

Flexibilidad limitada

A pesar de las considerables ventajas que en el campo de la flexibilidad ofrecen las arquitecturas microkernel frente al diseño monolítico de los sistemas operativos, todavía son muy reducidos los mecanismos que ofrece.

La modularidad interna de un sistema operativo diseñado con una arquitectura microkernel es una idea orientada fundamentalmente a conseguir aislar al sistema de las dependencias de la máquina, consiguiendo otra de las características deseadas, que era la portabilidad.

Sin embargo, los mecanismos soportados a más bajo nivel suelen ser estáticos y no es posible alterarlos en respuesta a características necesarias para aplicaciones específicas.

Menor Rendimiento frente a los sistemas operativos monolíticos

Los sistemas con micronúcleo tienen un rendimiento peor que los monolíticos, debido a que el paso de mensajes es más lento que la llamada directa a procedimientos. Sin embargo, trabajos como [Lie95], muestran cómo la pérdida de eficiencia y la escasa flexibilidad de los micronúcleos deriva, fundamentalmente, de una implantación inapropiada por lo que, con un diseño optimizado del mecanismo de comunicación pueden conseguirse muy buenos resultados.

Además, la pérdida moderada de rendimiento que se produce, es un hándicap que los usuarios pueden estar dispuestos a afrontar, frente a las ventajas que ofrece. Este hecho se produce también en otros casos, como en el caso de la arquitectura reflectiva.

2 Sistemas Operativos específicos de las aplicaciones

El término **sistema operativo específico de una aplicación**, del inglés *application-specific operating systems*, fue acuñado por [And92].

Surge como respuesta a una necesidad de mayor control del hardware por parte de las aplicaciones, ante la pérdida de rendimiento que, en algunos casos, puede suponer el hecho de que una capa intermedia como el Sistema Operativo, gestione los recursos físicos en nombre de las aplicaciones, pero no siempre de acuerdo a sus necesidades.

De ello se deriva un entorno de computación flexible, en el sentido de adaptable a los requerimientos de las aplicaciones y extensible.

Un ejemplo claro de tal problema son las bases de datos que, tradicionalmente, se ven obligadas a reimplementar partes del sistema operativo como el sistema de Entrada/Salida.

Como resultado, algunos trabajos han llegado a sugerir que lo mejor es "desenchufar" todo o partes del Sistema Operativo[ALK+90, Bla90] aunque, evidentemente, esto provocaría la pérdida de todas aquellas ventajas que hace años llevaron a los informáticos de la época a desarrollar sistemas operativos, como el contar con una plataforma de más alto nivel y, por tanto, más fácil de programar, sobre la que ejecutar las aplicaciones. Trabajos más extremistas como [EK95] proponen exterminar el sistema operativo.

El objetivo fundamental de un sistema operativo de este tipo es conseguir un alto rendimiento para lo que sugiere que se deje en manos de las aplicaciones conseguir el rendimiento del hardware, aunque, eso sí, ofreciéndoles, por un lado, las facilidades necesarias para compartir datos y recursos entre las aplicaciones y, por otro, un modelo de programación sencillo como el que se encuentra en un sistema operativo de propósito general. Todo esto sin requerir que el programador de la aplicación tenga que reescribir grandes trozos del sistema operativo.

De este tipo, los ejemplos más destacables son ExOS/Aegis[EKT95] y Panda[ABB+93]. Spin[BCE+94] está a medio camino entre una estructura de sistema operativo específico para cada aplicación y un microkernel tradicional.

Aegis[EKT95] es un sistema Operativo que obtiene flexibilidad con un buen rendimiento exponiendo las primitivas hardware de bajo nivel. Para ello, implementa un mecanismo de redirección de interrupciones – del inglés *traps* – para exportar servicios del hardware, como la gestión de excepciones o la gestión de la TLB, a las aplicaciones.

El sistema propiamente dicho, no ofrece ninguna abstracción más allá del conjunto de abstracciones mínimo proporcionado por el hardware. En lugar de esto, los servicios que tradicionalmente eran realizados por los sistemas operativos, como memoria virtual o planificación, se implementan como librerías que se ejecutan en el espacio de direcciones de las aplicaciones lo que permite que pueda cambiarse de acuerdo a las necesidades de la propia aplicación.

SPIN[BCE+94, BSP+95] construye un microkernel que permite a las aplicaciones tomar decisiones acerca de la política a implementar permitiendo que descargue extensiones de código en el kernel. Proporciona un pequeño conjunto de servicios en el núcleo a partir de los cuales se pueden construir otros servicios más sofisticados orientados a cada aplicación. Sin embargo, estos servicios se implementan como extensiones al propio kernel en lugar de cómo librerías.

Ninguna de ellas es OO, lo que resulta inadecuado a la hora de construir un sistema de soporte a objetos, aunque en el caso de SPIN el uso de lenguajes OO pueda ser de utilidad al definir módulos que extiendan el sistema, lo que hace difícil lograr aspectos como la uniformidad en todo el sistema, en torno a una abstracción como el objeto.

2.1 Organización

Se basa en que la mayor parte del sistema operativo se proporcione como servidores a nivel usuario o librerías que se enlazarían con cada aplicación, mientras que el trabajo de lo que se consideraría el sistema operativo se vería reducido a servir las peticiones de recursos físicos por parte de las aplicaciones y asegurar la protección necesaria entre las aplicaciones, sin definir ninguna abstracción. Este módulo básico, aunque mínimo, permanecería inalterable.

Cualquier otra cosa, desde la gestión de recursos hasta la comunicación entre aplicaciones, debe llevarse a cabo por medio de código del sistema operativo que se ejecuta como rutinas de librería en cada aplicación.

Evolución de los micronúcleos

En cierto sentido, se puede ver como una evolución natural de los microkernels. En este contexto, la principal innovación es que el kernel del sistema operativo proporciona interfaces que exponen la gestión de los recursos físicos a la aplicación en lugar de proporcionar interfaces de alto nivel como en los microkernels tradicionales.

Notificación de Eventos

El sistema operativo debe notificar a cada aplicación cualquier cambio que se produzca en su asignación de recursos, para permitir de esta forma, que las aplicaciones tengan la oportunidad de adaptarse de acuerdo a este cambio, siempre tratando de hacer el mejor uso de aquellos recursos que tenga disponibles.

Interfaz tradicional

Migrar la funcionalidad del sistema operativo al nivel de la aplicación no debe complicar la tarea del programador de aplicaciones. Para ello, la librería enlazada con cada aplicación es la responsable de proporcionar al programador de las aplicaciones la interfaz tradicional de cualquier sistema operativo.

Aplicación de Tecnologías Orientadas a Objeto

Se trata de construir el sistema operativo fuera del kernel. Para ello, se aconseja utilizar técnicas de orientación a objetos para proporcionar implementaciones personalizables de los componentes a nivel usuario del sistema.

2.2 Ventajas e Inconvenientes

2.2.1 Ventajas

Fundamentalmente, son dos las ventajas derivadas de este método de estructuración de un Sistema Operativo.

Adaptabilidad y Extensibilidad dinámicas

Esta forma de estructura un Sistema Operativo permite un alto grado de adaptabilidad de los sistemas operativos por parte de las aplicaciones. Es más, son las propias aplicaciones las que se ocupan de proporcionar las abstracciones de alto nivel del Sistema Operativo, adaptándolas de acuerdo a sus necesidades.

Rendimiento

Este tipo de Sistemas Operativos permiten que cada aplicación defina sus propias abstracciones de alto nivel, adaptándolas a los requerimientos de la aplicación y gestionando así de la forma más eficiente posible los recursos del sistema.

2.2.2 Inconvenientes

Los inconvenientes de este método de estructuración dependen en gran medida de la implantación concreta y, fundamentalmente, de la forma en que el sistema operativo permita la intervención de las aplicaciones.

Sin embargo, tales intervenciones suelen estar caracterizadas por ser **altamente restrictivas** y **difíciles de llevar a cabo**. Por ejemplo, el mecanismo la gestión de eventos en SPIN ha sido criticado por ser muy complejo[CD94] y las extensiones del núcleo del sistema operativo deben ser compiladas por un compilador de confianza.

3 Familias de Programas

El diseño de un sistema operativo basado en el uso de **familias de programas** fue propuesto a principios de los 70 en [HFC76]. Sin embargo, esta propuesta fue ignorada por los investigadores en el campo de los sistemas operativos hasta que las técnicas de OO fueron ampliamente aceptadas en el campo de los sistemas operativos a finales de los 80[CS91].

Hoy en día, el miembro más representativo de esta corriente de diseño es PEACE[SPr94] sin olvidar a Choices[CIM92].

3.1 Definición

Una **familia de programas** se define como un conjunto de programas cuyas propiedades comunes son tan extensas que es más ventajoso estudiar ese conjunto de propiedades comunes antes de analizar los elementos individuales[Par76].

Es posible que los miembros de una familia se diferencien únicamente en que la configuración hardware en que se ejecutan, o que algunos miembros de la familia proporcionan únicamente un subconjunto de las características de los otros miembros, tal vez porque sus clientes no necesitan el resto de las características o porque un miembro concreto decide implementar ciertas características de forma diferente, en base a la funcionalidad que las aplicaciones le van a exigir.

En ningún caso esta aproximación a la construcción de Sistemas Operativos determina una técnica de implantación particular aunque al final de este apartado se hará un estudio de cómo el concepto de familia de programas encaja especialmente bien con las tecnologías OO.

3.2 Organización

El diseño de una familia de programas distingue entre un **subconjunto mínimo de funciones del sistema** también llamado **base mínima** y un conjunto de **extensiones mínimas del sistema**[SPr93b].

Subconjunto Mínimo de Funciones del Sistema

Define una plataforma básica que proporciona las abstracciones fundamentales útiles para implementar las extensiones sobre ella.

Es posible que este subconjunto mínimo sea suficiente para las necesidades de algunos clientes.

Extensiones Mínimas del Sistema

El resto de los miembros de la familia se implementan construyendo extensiones mínimas que proporcionan funcionalidad adicional al sistema.

Siguiendo las bases de un **diseño incremental**[HFC76] cada nuevo nivel que se construye sobre la base mínima, que supone una nueva extensión, define una máquina abstracta que constituye una nueva base mínima para nuevas extensiones de más alto nivel.

3.2.1 Diseño guiado por la aplicación

Las extensiones se realizan únicamente bajo demanda, es decir, cuando sea necesario para implementar una característica específica del sistema que soporte una aplicación determinada.

Así, el diseño de sistemas operativos basados en familias de programas, pueden aplicar el principio de **guiado por la aplicación**.

Llevado a sus últimas consecuencias, las aplicaciones llegarían a ser las extensiones finales del sistema, desapareciendo la tradicional barrera entre aplicaciones y Sistema Operativo. Las aplicaciones extenderían al Sistema Operativo y este complementaría a aquellas.

3.2.2 Orientación a Objetos

Dada la fuerte analogía entre las nociones de familia de programas y OO (Ver Figura 2.4), es casi inmediato construir familias de programas usando un marco OO[SPr93a].

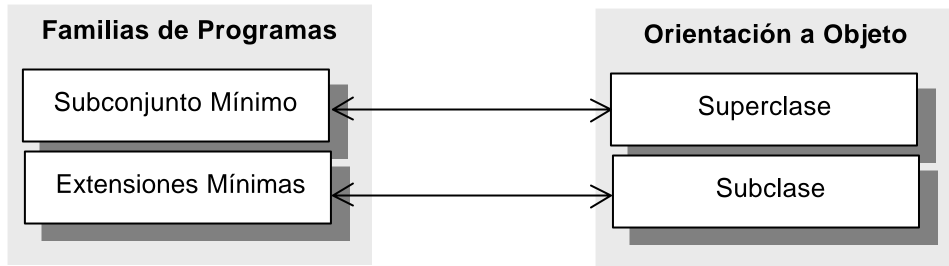


Figura 2.4 Familias de Programa versus OO

El subconjunto mínimo de funciones del sistema en el concepto de familia de programa, tiene su contrapartida en la superclase de la aproximación OO. Las extensiones mínimas son introducidas por medio de subclases.

Herencia y polimorfismo permiten la coexistencia de diferentes implementaciones con la misma interfaz.

La aplicación de la OO como mecanismo de implantación del sistema operativo suma varias ventajas a las muchas que de por sí ofrece el principio de familia de programas. De esta forma, la herencia facilita la implantación y el mantenimiento de un sistema incremental. La reutilización del código se mejora sensiblemente lo que no es nada despreciable si se recuerda la definición de familia de programas. Por último, la aplicación de técnicas OO en la implantación, refleja adecuadamente la modularidad que la aplicación del concepto de familia de programas conlleva como técnica de diseño.

3.3 Ventajas e Inconvenientes

3.3.1 Ventajas

La aplicación del concepto de familia de programas al diseño de un Sistema operativo proporciona una serie de ventajas [SPr95].

Estructura Modular

Representar un Sistema Operativo como una familia de programas que se extienden incrementalmente, en el que las nuevas características del sistema se añaden a un conjunto ya determinado de funciones del sistema, dota al sistema de una estructura altamente modular.

Adaptación

La principal ventaja de esta estructura es el soporte que ofrece para la adaptación de grano fino.

Una vez identificado el subconjunto mínimo de funciones del sistema, las aplicaciones pueden ajustar el comportamiento del sistema operativo mediante

extensiones que se construyen sobre la base mínima, determinando de esta forma el comportamiento final del sistema.

Suponen por tanto la adaptación del sistema en base a la especialización del mismo para una aplicación específica, en lugar de construir un sistema para cualquier tipo de aplicación, como era habitual.

Poca Sobrecarga

Dado que las extensiones son guiadas por las aplicaciones, los componentes del sistema se ajustan siempre a una aplicación específica. Estos componentes introducirán sobrecarga sólo cuando intenten proyectar una aplicación específica en la máquina abstracta de más bajo nivel. En cuyo caso, la sobrecarga en el proceso de correspondencia será la misma que si esta correspondencia fuese hecha por la propia aplicación.

3.3.2 Inconvenientes

Flexibilidad Estática

La mayor carencia de este tipo de sistemas es la falta de flexibilidad dinámica. Proporcionan únicamente flexibilidad estática, lo que permite adaptar el sistema a las necesidades de una aplicación o conjunto de aplicaciones en tiempo de construcción del sistema, ya sea en la compilación, el enlace o la carga. Sin embargo, una vez arrancado el sistema, no puede adaptarse a nuevos requerimientos[Sch96].

4 Orientación a Objetos

Las técnicas de desarrollo software Orientadas a Objeto han sido y son utilizadas de forma habitual en el nivel de aplicación y, desde finales de los años 80, se vienen empleando también como herramienta de diseño y construcción de sistemas operativos[FHG93, MCK91]. Aunque en el capítulo VI se verá una discusión más extensa de este tema, presentaremos a continuación algunas ideas acerca de la aplicación de las tecnologías OO al diseño de SO.

4.1 Definición

Aunque existen varios sistemas operativos, tanto micronucléos como sistemas operativos monolíticos, que han sido implementados utilizando lenguajes orientados a objetos como Choices[CIM92], la definición de sistema Operativo Orientado a Objetos es algo más que utilizar un lenguaje de programación Orientado a Objetos para su implantación.

Un **sistema operativo orientado a objetos** es aquel que no sólo está implementado usando técnicas de orientación a objetos sino que además, debe soportar el concepto de objeto como elemento con entidad propia, con una semántica fuerte y bien definida, proporcionando soporte tanto a los objetos del propio sistema como de usuario sin que deban tener características especiales, a la vez que ofrece sus servicios como invocaciones a objetos del sistema[Rus91a].

Son muchos los proyectos de SO que se catalogan como SOOO. Aunque, dependiendo del grado en que se ajustan a la definición anterior, pueden establecerse distintos tipos de SOOO.

Independencia del Lenguaje

Un SOOO no debe restringirse a un modelo de objetos o lenguaje particular, sino ofrecer los mecanismos genéricos de gestión de objetos de lo que se puede derivar que, cualquier Sistema Operativo que quiera recibir el nombre de SOOO debe[Sha91a]:

- Crear una plataforma para el desarrollo y ejecución de aplicaciones
- Facilitar la implantación de compiladores y entorno en tiempo de ejecución para lenguajes OO
- Crear una plataforma en tiempo de ejecución para soporte de objetos que podrán comunicarse, incluso aunque hayan sido definidos por diferentes aplicaciones.

4.2 Aproximaciones Orientadas a Objeto en los Sistemas Operativos

En el caso de la aplicación de las tecnologías OO a sistemas como los sistemas operativos o las bases de datos, el enfoque principal no está en la construcción de un sistema software grande y complejo aplicando técnicas de OO, lo cual no deja de ser relevante, dadas la gran cantidad de ventajas que de ello se derivan y de las que se hizo una breve reseña en el Capítulo I.

Dado que un sistema operativo es un sistema activo, que se ejecuta sobre una máquina hardware y que, a su vez, ofrece a las aplicaciones una máquina abstracta sobre la que ejecutarse, el principal enfoque está en el tipo de abstracciones que proporciona en tiempo de ejecución y el control que sobre ellas se puede ejercer para que realicen algún trabajo.

Por tanto, a grandes rasgos, se pueden destacar cuatro aproximaciones a los SO con OO[GKS94].

- **Aproximación del lenguaje**. La OO se aplica, únicamente, en la producción del código del sistema operativo, realizando un diseño OO y explotando para ello las características de un lenguaje de programación OO.

  Sin embargo, en tiempo de ejecución, el sistema no ofrece ninguna estructura OO, sino que se transforma en un sistema tradicional, con las abstracciones tradicionales. Después de compilar y enlazar el sistema, no existe ningún conocimiento acerca de clases, objetos o relaciones entre ellos.

- **Aproximación de gestor de objetos**. Algunos sistemas operativos se construyen como gestores de objetos, de forma que las aplicaciones OO son directamente soportadas por una infraestructura no OO, donde el lenguaje de implantación carece de importancia. Los sistemas operativos conocen el concepto de objeto y son capaces de gestionarlos, utilizándolos como las abstracciones básicas para proporcionar servicios, a la vez que dotan al sistema de características como la adaptación, migración de objetos, etc.

  Sin embargo, la estructura resultante difiere considerablemente de las estructuras utilizadas en el desarrollo. Además, en muchos casos, los objetos en tiempo de

ejecución representan abstracciones tradicionales como los procesos o los espacios de direcciones, muy pesados, eliminando la posibilidad de adaptación de grano fino. Hay dos posibilidades:

a) Sistema Operativo ejecutándose sobre una máquina hardware

b) Extensiones a un sistema operativo tradicional en tiempo de ejecución

- **Aproximación cliente/servidor en micronúcleos**. Los procesos servidores son instancias que proporcionan un servicio (objetos) y el micronúcleo constituye la infraestructura básica. La funcionalidad del sistema operativo se proporciona por procesos servidores.
- **Aproximación multinivel**. Para cubrir el salto entre el desarrollo del sistema y el sistema en tiempo de ejecución, es necesario retener toda la información estructural acerca de las clases, los objetos, y las relaciones entre ellos. Es más, es necesario asignar la funcionalidad, que actualmente lleva a cabo un entorno en tiempo de ejecución (por ejemplo, la creación y destrucción de objetos), a componentes (objetos) del sistema identificables. De esta forma se consigue integrar los niveles de aplicación y sistema en una arquitectura homogénea a la vez que se dota al sistema de adaptabilidad dinámica llevando a cabo las mismas modificaciones y extensiones en el sistema en ejecución[Sch96].

4.3 Ventajas de la estructuración con objetos del sistema.

Los mecanismos de OO son ampliamente aceptados como un poderoso método para modelar y estructurar sistemas. Tales mecanismos, simples y eficientes, como la encapsulación, la herencia y el polimorfismo, proporcionan ventajas significativas como su tendencia a producir sistemas bien estructurados, un alto grado de reusabilidad o adaptabilidad por medio de la aplicación de la herencia, por citar algunos[Rus91b, Cah96a].

Añadidas a estas, el uso de técnicas de orientación a objetos para diseñar un sistema operativo, dota al sistema de flexibilidad, proporcionando la posibilidad de construir sistemas altamente personalizables[Kra93].

A continuación, se presentan algunas de las ventajas del diseño de SO aplicando Tecnología OO. En el Capítulo VI se presentarán nuevas ventajas derivadas del desarrollo de un SOOO en tiempo de ejecución, como base para el soporte de otras capas del sistema OO.

4.3.1 Reutilización.

La mayor parte de los Sistemas Operativos contienen partes dependientes de la arquitectura de la máquina que deben reescribirse para cada arquitectura en que se quiera implantar.

Los sistemas operativos construidos según un método tradicional, normalmente, especifican tales módulos en función de las operaciones que proporcionan. Sin embargo, sin la herencia, implantaciones similares de una misma interfaz serán independientes, sin ninguna posibilidad de reutilizar o compartir código común.

El paginador externo del sistema operativo MACH, **pmap**[RTY+87] es un ejemplo. Por su naturaleza dependiente de la máquina, las operaciones que componen el sistema pmap debe ser reimplementadas, o al menos, modificadas, para cada arquitectura en la que se quiera instalar Mach. Las implementaciones para estas arquitecturas similares no tienen ningún medio de compartir código.

Por otro lado, los SO normalmente, ofrecen estructuras de datos similares para implementar varias políticas y mecanismos entre versiones de un SO o dentro de una versión. Esto también se verá beneficiado de la reutilización proporcionada por la herencia.

Por ejemplo, el sistema de ficheros de UNIX gira en torno al concepto de i-nodo [Bac86]. System-V[ATT91] y BSD[MBK+96] implementan los i-nodos de diferentes modo manteniendo, sin embargo, una semántica y comportamiento idénticos.

El soporte para tales situaciones lo proporcionan las tecnologías OO por medio de la herencia, construyendo clases abstractas que definen la interfaz y localizan el comportamiento común y utilizando subclases concretas que localizan las diferencias.[referencia russomlrc88 y mad 91].

4.3.2 Portabilidad

Para diseñar un Sistema Operativo portable, es necesario aislar las dependencias de las características particulares de los dispositivos o arquitectura hardware subyacente en módulos separados y opacos al resto del sistema.

Los primeros constituyen lo que se ha dado en llamar **módulos dependientes de la arquitectura** y proporcionan una interfaz para el acceso a la entidad que abstraen y encapsulan, ocultando los detalles concretos y permitiendo diseñar e implementar el resto del sistema, **módulos independientes de la arquitectura**, de forma que sólo dependan de la interfaz, no de la implantación concreta.

Idealmente, tales módulos podrían ser reescritos en caso de que sea necesario migrar a una arquitectura distinta, sin modificaciones en las partes independientes de la arquitectura.

Las TOO proporcionan un marco adecuado para dar soporte a tales necesidades de encapsulación. Se pueden utilizar clases abstractas para definir las interfaces de los módulos dependientes de la arquitectura, a la vez que se utilizan clases concretas para implementar estas interfaces para arquitecturas concretas. El polimorfismo permite utilizar una instancia de cualquiera de las clases concretas definidas.

4.3.3 Separación Interfaz/Implantación

Cualquier intento de diseñar un sistema operativo extensible, pasa por separar las políticas de los mecanismos que las implementan[SG98].

Esto hace necesario identificar y aislar aquellas políticas con interfaz similar, lo que permite poner en práctica distintas políticas a medida que cambien los requerimientos, simplemente reemplazando un módulo que implementa una política particular por otro.

Las TOO permiten representar las políticas con interfaz común por medio de clases abstractas que definen tal interfaz. Clases concretas pueden implementar la interfaz para políticas específicas.

Un buen ejemplo es la planificación en un sistema multiprogramado. Aplicar las TOO supone crear una clase abstracta Planificador que encapsula las propiedades comunes a cualquier planificador, como la cola de procesos listos, y define la interfaz común a cualquier planificador como **encolar** y **encontrar_el_siguiente**.

Cada subclase proporciona la misma interfaz pero implementa los métodos encolar y encontrar_el_siguiente de acuerdo a la política que defina.

4.3.4 Adaptabilidad

La separación real que el uso de la OO permite establecer entre la interfaz y la implantación de un servicio, junto con mecanismos inherentes a la tecnología OO como la herencia permite, no sólo ofrecer diferentes implementaciones de una misma interfaz de un servicio, sino que todas coexistan pacíficamente, e incluso, es posible llegar a permitir que una aplicación proporcione su propia implantación a un servicio del sistema.

Esto desdibuja la frontera tradicionalmente existente entre servicios de la aplicación y del sistema y permite a un usuario instalar módulos con servicios del sistema e instalarlos, siempre y cuando se adapten a la interfaz requerida[HKN91].

Así, se proporciona un sistema adaptable a necesidades específicas. Se puede citar como ejemplo al sistema operativo Choices[CIM+93], basados en frameworks, que soportan la reutilización del código y la adaptación a través de la herencia[CIM92].

Un paso más consiste en exponer las clases del sistema a las aplicaciones de forma que puedan aprovecharse de la herencia y el polimorfismo para proporcionar implementaciones especiales de algunos servicios del sistema que se adapten a sus necesidades. Un claro exponente es PANDA[ABB+93] y algunas versiones avanzadas de Choices[LTS+96].

4.3.5 Facilidad para evolucionar

Los sistemas estructurados en términos de objetos permiten reemplazar partes del mismos por lo que evolucionan fácilmente. Además, es posible reutilizar y adaptar a objetos e incluso subsistemas a nuevas condiciones del entorno gracias nuevamente a mecanismos como la herencia.

4.3.6 Optimización Estructurada

Dado que la herencia permite la especialización, al tiempo que se consigue reutilizar y compartir código, la aplicación de la misma junto con el polimorfismo, da lugar a una nueva ventaja conocida como optimización estructurada.

En un sistema operativo, los algoritmos son constantemente refinados y optimizados. La utilización de TOO proporciona el marco adecuado para soportar las optimizaciones necesarias. Clases abstractas pueden definir interfaces que, por razones de rendimiento, pueden ser implementadas por clases concretas que imponen diferentes restricciones para su uso.

5 Reflectividad: Implantación Abierta

El modelo de abstracción tradicional ofrecido por un Sistema Operativo, basado en la ocultación de información en una caja negra, protege a los clientes de una abstracción de tener que conocer los detalles de cómo está implementada esta abstracción. Sin embargo, este modelo tradicional, no sólo eleva la visión del usuario sino que por otro lado evita que pueda cambiar los detalles de implantación incluso cuando lo desea por cuestiones importantes[KLM+93].

5.1 Necesidad de una Implantación Abierta: Dilemas, Conflictos y Decisiones

Algunos detalles de implantación que, tradicionalmente, se han ocultado detrás de las abstracciones ofrecidas por el sistema operativo, son algo más que meros detalles. Algunos son estrategias de implantación cuya resolución variará sin remisión la funcionalidad o el rendimiento de la implantación resultante [KL93].

A estos detalles se les denomina **dilemas de correspondencia**, del inglés, *mapping dilemmas*, y las decisiones de cómo resolverlo **decisiones de correspondencia**, del inglés, *mapping decisions*.

Cuando un cliente determinado de un servicio tiene un rendimiento pobre debido a que la implantación de ese servicio oculta una decisiones de correspondencia inapropiadas para ese cliente concreto, estaremos ante un **conflicto de correspondencia**, del inglés, *mapping conflict*.

Estos términos fueron acuñados por G.Kiczales en su trabajo[KLM+93].

Dilemas, Conflictos y Decisiones en el Sistema Operativo

A medida que el hardware ganaba en complejidad, los sistemas Operativos se han encontrado con dilemas cada vez más intrincados, debido, fundamentalmente a que las decisiones que se tomen tendrán mayores consecuencias.

El objetivo es mantener las abstracciones que han dado a los Sistemas Operativos el éxito de que gozan, pero asegurando, de alguna forma, que la implantación concreta de cada abstracción es la más apropiada para cada caso[KLM+93].

Se hace necesaria la participación del usuario (desarrollador de aplicaciones, fundamentalmente) en la toma de decisiones. Sin embargo, esto no supone que se quiera programar a bajo nivel, sino que se quiere aprovechar las ventajas que las abstracciones de alto nivel ofrecen, teniendo a la vez la oportunidad de señalar cómo se implementan esas abstracciones.

5.2 El Modelo de Implantación Abierta

El modelo de implantación abierta[KTW92] propone que los clientes de una abstracción debería ser capaces de ejercer cierto control sobre decisiones tomadas en la implantación de tal abstracción para adecuarla mejor al uso que se pretende darle.

La cuestión fundamental es cómo hacerlo y cómo hacerlo bien. No hay que olvidar que se desea evitar enterrar a los usuarios en una maraña de detalles o limitar a los implementadores del servicio.

La idea original de los Sistemas Operativos era controlar la complejidad exponiendo la funcionalidad y ocultado la implementación.

La idea fundamental de la aproximación abierta es controlar la complejidad separando el acceso del cliente a la funcionalidad que proporciona un servicio, del acceso del cliente a las decisiones de correspondencia de ese servicio. Es decir, el modelo de implantación abierta sigue una aproximación de **separar, en lugar de ocultar**[KL93].

La diferencia fundamental entre ambas estriba entonces en que, mientras la primera, basada en la ocultación de detalles de implantación detrás de una abstracción, expone la funcionalidad pero oculta los detalles de implantación, la segunda se basa en controlar la complejidad separando el acceso del cliente a la funcionalidad y a la implantación del servicio.

El modelo de implantación abierta fue la consecuencia del trabajo realizado en reflectividad y, en concreto en el uso de protocolos meta-objetos en la implantación de LPOO [KRB91].

5.2.1 Interfaz base versus Meta Interfaz

La separación de acceso a la funcionalidad/acceso a la implantación de un servicio promovida por la implantación abierta, propicia una división de la interfaz que presenta un servicio en dos, denominadas **interfaz base** y **meta interfaz** o **protocolo base** y **meta-protocolo**.

El modelo de implantación abierta se basa en la diferencia entre la interfaz base y la meta-interfaz de una abstracción[KL93] para ofrecer a los clientes de una abstracción la posibilidad de influir en su implantación evitando la necesidad de conocer excesivos detalles.

Interfaz base de una abstracción

Ofrece la funcionalidad usual de la abstracción. Los clientes del servicio escriben sus programas sobre esta interfaz base del modo tradicional.

Meta Interfaz de una abstracción

Permite a los clientes controlar cómo se implementan ciertos aspectos de la abstracción. Es pues, el medio para exponer a los clientes, de forma controlada, los detalles de implantación de una abstracción. Proporciona una visión abstracta de la implantación de la abstracción de forma que los clientes puedan ajustarla a sus necesidades[KL93].

5.2.2 Ventajas de la división nivel base / meta nivel

Existen múltiples ventajas derivadas del uso de arquitecturas con soporte explícito para la meta-programación[KL93]. La separación interfaz base versus meta-interfaz se ha manifestado como uno de los medios más potentes de proporcionar flexibilidad a un sistema software. Por ejemplo, la interfaz del paginador externo de Mach se puede ver como la meta-interfaz al sistema de memoria virtual de Mach.

Incrementalidad

Si un cliente quiere cambiar una pequeña parte de la implantación de un servicio, será necesario escribir un meta-programa, proporcionalmente pequeño. Así, en la implantación de un sistema operativo tradicional utilizando técnicas como las aquí descritas, si un cliente quiere una política de reemplazo de páginas diferente, debería describir la nueva política, no todo el mecanismo.

Ámbito de control

Los cambios hechos a través de una meta-interfaz afectarán sólo a algunas entidades del programa base. Así, puede haber distintas políticas de reemplazo de páginas en distintas partes de un mismo cliente.

Interoperabilidad

Aquellos trozos de un programa base que sigan políticas no estándar pueden interactuar con aquellas otras partes del programa base que utilicen las políticas implementadas por defecto.

5.3 Sistemas Reflectivos

El uso de meta-protocolos o meta-interfaces es un resultado de trabajos previos en la reflectividad y la orientación a objetos.

La combinación de los **Meta-Protocolos** y la orientación a objetos da como resultado los *meta-object protocols* o **MOP** que, esencialmente, representan una **Meta-interfaz** a la implantación del modelo de objetos de un lenguaje, lo que permite al programador de aplicaciones controlar cómo se implementan ciertos aspectos del modelo de objetos como la invocación de métodos en objetos, el despacho de métodos o la herencia, entre otros[KL93, GC96].

Un **sistema reflectivo** es aquel que está implementado en función de sí mismo[Mae88]. Por ejemplo, un lenguaje reflectivo es aquel en el que la implantación del lenguaje usa el propio lenguaje.

En el capítulo XII se hace un tratamiento más extenso de este tipo de sistemas, clave en el desarrollo de esta tesis.

Trabajos paralelos han aplicado las mismas ideas a SOOO buscando el desarrollo de SOOO reflectivos. El trabajo más destacable es Apertos [Yok92].

5.4 Ventajas e Inconvenientes de estos sistemas

5.4.1 Ventajas

Los sistemas reflectivos son, fundamentalmente muy flexibles, ofreciendo, además, una flexibilidad de grano fino. Esto hace que los sistemas reflectivos sean altamente configurables y adaptables.

5.4.2 Inconvenientes

Por contrapartida, ofrece un rendimiento menor que los sistemas en los que no se expone la implantación. Sin embargo, como ya se comentaba en el apartado dedicado a los micronúcleos, puede ser una merma soportable, frente a las ventajas que proporciona. Por otro lado, la investigación en técnicas como la generación dinámica de código puede aliviar estos problemas en el futuro.

6 OO + reflectividad: una combinación prometedora

La utilización de técnicas de OO favorecen la adaptabilidad de los sistemas. La idea fundamental en la utilización de las técnicas de OO, es agrupar las distintas partes de un sistema en objetos y poder especificar diferentes decisiones para funciones particulares ofreciendo una determinada clase de objetos que implemente esa determinada decisión[KL93].

De tal forma que sea posible realizar adaptaciones del mismo para cada objeto o conjunto de objetos que deseen un comportamiento particular con el propósito de lograr un mayor rendimiento o un menor riesgo de la consistencia del estado interno del objeto.

Capítulo III

Entornos de Computación definidos por los Sistemas Operativos

La construcción de un Sistema Operativo OO, que ya en el capítulo anterior destaca como una solución al problema de la interoperabilidad, implicará el soporte de la abstracción de objeto como una entidad identificable que proporciona servicios a sus clientes, algo que se identificó como una característica *sine qua non* en el Capítulo I. El servicio en sí lo lleva a cabo lo que se denominará **actividad**, la representación de la computación en el sistema.

Como parte del diseño de este nuevo sistema operativo, será necesario definir el entorno de computación de un Sistema Operativo que soportará el concepto de objeto. Se verá que, lo que semeja una tarea sencilla como es introducir la concurrencia en el entorno de computación, hace peligrar propiedades tan elementales como la encapsulación.

Esto plantea dos cuestiones fundamentales. Por un lado, determinar qué clase de abstracciones ofrecerá el sistema operativo para representar las distintas actividades que se llevan a cabo en el sistema, el modelo de concurrencia soportado y la interacción entre ellas. Por otro, qué tipo de relación existirá entre las actividades y los objetos.

En este capítulo, se hará un breve repaso a los entornos existentes, varios de ellos ampliamente difundidos y de sobra conocidos. En capítulos posteriores se elegirá un modelo que se analizará más detenidamente.

1 Necesidad de un entorno de computación adecuado para un sistema Orientado a objetos

Una de las tareas tradicionales de cualquier sistema operativo ha sido la de ofrecer un entorno de computación en el que dar soporte a la ejecución de las aplicaciones tanto de los usuarios como del resto de los componentes del sistema operativo[Sta98].

Con ello, proporciona al usuario un entorno **cómodo** para la ejecución de sus tareas, **eficiente**, al administrar los recursos intentando el máximo aprovechamiento de los mismos y, a ser posible, con **capacidad para evolucionar** a medida que cambian los requerimientos de las aplicaciones de usuario.

1.1 El entorno de computación como aislante del hardware

El entorno de computación del sistema operativo constituye un entorno amigable, al menos, mucho más que el hardware, que aísla al usuario del hardware amortiguando la enorme distancia conceptual existente entre ambos (usuario y hardware subyacente), a la vez que le ofrece un conjunto de servicios básicos accesibles a través de una interfaz, conceptualmente, cercana al usuario[Sta98].

Este entorno debía ser potente para extraer la máxima potencia de la máquina y que a la vez ofreciese abstracciones sencillas de forma que resultase simple y comprensible a los programadores.

La integración de las Tecnologías Orientadas a Objeto a la hora de diseñar el Sistema de Computación de un Sistema operativo es, sin embargo, algo oscuro y complejo.

Algunos sistemas operativos orientados a objeto como Choices[RJC88] aprovechan las tecnologías orientadas a objetos para encapsular cada elemento hardware en un objeto y poder así integrarlo de forma homogénea con el resto del sistema.

Sin embargo, esta aproximación no supone la integración del concepto de objeto en el Sistema Operativo, sino la utilización, como herramienta de diseño, de la TOO para la construcción de una aplicación grande y compleja como es un SO.

1.1.1 El entorno de computación como proveedor de abstracciones

Históricamente, los sistemas operativos han ofrecido modelos de computación basados fundamentalmente en las abstracciones de proceso, tarea (del inglés, *task*), hilo o hebra (del inglés, *thread*), etc. Estas abstracciones constituían las unidades básicas de ejecución, englobando todos aquellos recursos necesarios para ejecutarse. Además, el Sistema Operativo también proporcionaba algunos mecanismos para comunicarlas y sincronizarlas.

Hay multitud de trabajos acerca del tema. Es más, todos los libros clásicos de sistemas operativos lo tratan [TW98, Bir91, Sta98, Bac92, Vah96].

La abstracción paranoica

El origen de todas estas abstracciones nace en el hecho de que los sistemas tradicionales usan lenguajes inseguros como el C, de tal forma que todos los programas que se escriben son intrínsecamente inseguros. Esto obliga a envolverlos (del inglés, *wrapped*) en algún marco de protección para que sean seguros. Este marco es el proceso, tarea, etc.

Algunos autores como [~Tun] califican a este marco de **abstracción paranoica** y afirman que son un lastre de los Sistemas Operativos tradicionales que hay que eliminar. En estos trabajos se propugna el **liberalismo computacional**, del inglés, *computing liberalism*, que propone eliminar las abstracciones de grano grueso como el proceso.

Como se verá a lo largo de este capítulo, las abstracciones ofrecidas por los sistemas operativos tradicionales para representar la computación no son adecuadas para un sistema orientado a objetos. El modelo de objetos soportado por el sistema determinará

el modo de integración de la computación en el sistema. Este aspecto se discute en el capítulo X.

Mecanismos para definir el nivel de Concurrencia

Los sistemas operativos también ofrecen los servicios necesarios para introducir la concurrencia[Sta98], como pueden ser: clonación y destrucción de procesos dotándolos de vida independiente, mecanismos de planificación de los distintos procesos existentes y mecanismos que permitiesen la sincronización de las entidades de ejecución en puntos determinados[Ben90], fundamentalmente, en el acceso a las entidades de almacenamiento de datos. Se pueden citar los semáforos como los mecanismos que alcanzaron mayor popularidad.

Sobre estos sistemas operativos, el sistema en tiempo de ejecución de los lenguajes no concurrentes, es decir, aquellos que no disponían de un soporte en tiempo de ejecución que soportase construcciones de concurrencia, podían utilizar los mecanismos ofrecidos por el sistema operativo.

Por ejemplo, algunos lenguajes como Módula-2[Wir90] proporcionan corutinas para expresar subprogramas independientes dentro del mismo programa permitiendo tanto compartir datos como definir datos privados.

El lenguaje proporciona instrucciones para crear y destruir corutinas y para permitir a una corutina suspender temporalmente su ejecución pero reteniendo su estado [Bac92].

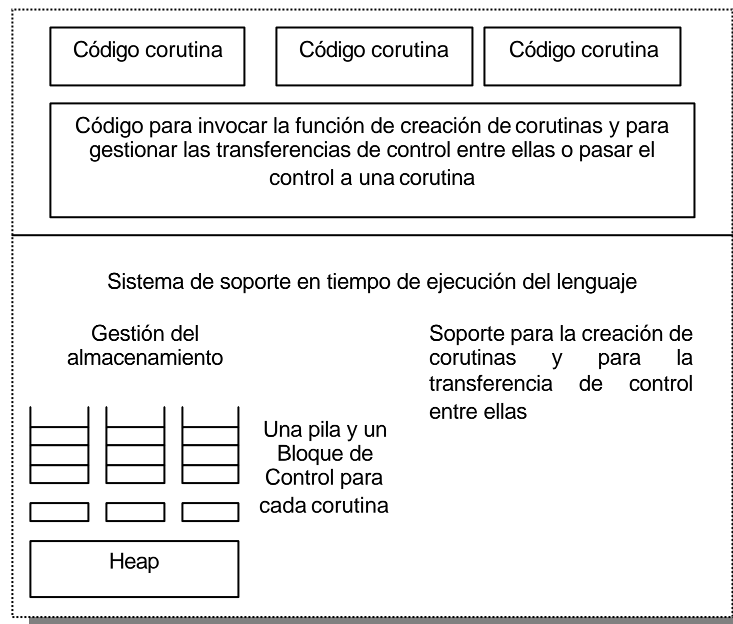


Figura 3.1 Soporte del lenguaje para corutinas

Se han desarrollado multitud de combinaciones al introducir modelos de concurrencia, iguales o diferentes, a nivel de sistema operativo y lenguaje.

En cualquier caso, por debajo de todo esto, el soporte lo daba una máquina hardware que ofrecía un modelo de computación muy diferente al ofrecido por lenguaje + sistema operativo. Mientras estos ofrecen una visión de procesos en ejecución sincronizándose

con algún mecanismo de más o menos alto nivel, la máquina ofrece una visión basada en regiones de memoria, registros y pila y un conjunto de instrucciones de muy bajo nivel para cargar, almacenar y/o transferir datos entre memoria y registros y ejecutar la siguiente instrucción señalada por un determinado registro. Además, con el fin de facilitar la ejecución de actividades concurrentes, la máquina hardware subyacente puede ofrecer instrucciones que permitan habilitar o deshabilitar interrupciones para evitar cualquier interferencia en la ejecución de determinadas operaciones críticas.

1.2 El sistema operativo como gestor de recursos

Además de elevar el nivel de la máquina ofreciendo al usuario una máquina virtual más sencilla de programar a través de un conjunto de abstracciones de más alto nivel, el Sistema Operativo también debe ofrecer los mecanismos necesarios para gestionar tales abstracciones, encapsulación de los recursos de la máquina.

La gestión debe asegurar, no sólo la consistencia de los recursos, sino también la eficiencia global del sistema.

Así, el sistema operativo decide quién, cuándo y con qué recursos se lleva a cabo cada tarea, para lo que debe tener un conocimiento total del estado del sistema.

2 Requisitos de Diseño de un Entorno de computación

La bondad de un sistema se mide no sólo por la funcionalidad que ofrece, sino también por la fidelidad a unos principios de diseño generales que miden el grado de adecuación del mismo. Los objetivos de diseño pueden entonces catalogarse en dos vertientes distintas. Los requisitos funcionales y los no funcionales [Vah96][WS95].

2.1 Requerimientos funcionales

Los requerimientos funcionales o características operacionales de que debe estar dotado el sistema de computación, se refieren al propósito del sistema, es decir, a lo que hace o debería hacer el sistema.

Tradicionalmente, estos requerimientos han sido, por un lado, elevar el nivel de abstracción de la máquina y, por otro, ofrecer mecanismos de gestión de la computación sencillos y potentes.

2.1.1 Abstracciones adecuadas

El sistema debe ofrecer abstracciones que representen el poder de computación de la máquina, a un nivel adecuado para el usuario. Con este objetivo, en este capítulo se lleva a cabo un rápido estudio de las abstracciones proporcionadas por los Sistemas Operativos tradicionales para representar la ejecución concluyendo, al final, su falta de adecuación para constituirse en soporte de un SOOO.

2.1.2 Concurrencia en el Modelo

La concurrencia o habilidad de ejecutar diversas acciones a un tiempo, se ha revelado como una característica importante en cualquier sistema operativo. La posibilidad de aprovechar los periodos de inactividad del hardware subyacente o la presencia de varios procesadores físicos, ha supuesto un avance hacia la ejecución eficiente de las aplicaciones de usuario.

El modelo de computación busca, ante todo, introducir la concurrencia en el sistema con el objetivo de mejorar el rendimiento por encima de los límites de la máquina subyacente. Estos intentos se han llevado a cabo de distintas formas con distintos enfoques y han tenido diferente éxito.

Determinadas las abstracciones ofrecidas por el sistema operativo, será también necesario determinar el nivel de concurrencia permitido en el modelo y cómo se puede expresar la concurrencia en el mismo.

2.1.3 Comunicación y sincronización

El modelado de un problema del mundo real mediante las abstracciones proporcionadas por el sistema operativo, trae consigo la necesidad de un mecanismo de comunicación que se integre con ellas permitiéndoles intercambiar información.

Además, la ejecución concurrente de varias actividades en el sistema, hace necesario algún mecanismo de sincronización que, al igual que el anterior, combine bien con las abstracciones definidas por el sistema operativo.

Tradicionalmente, en los entornos OO, se ofrecen mecanismos organizados a imagen y semejanza de los mecanismos de comunicación y sincronización tradicionales de forma similar a las llamadas a procedimientos locales o remotos que operan de forma síncrona.

Sin embargo, este tipo de mecanismos se manifiestan insuficientes ante los nuevos requerimientos de las aplicaciones (asincronismo, comunicación basada en envío de mensaje y no en llamada a procedimiento, etc.).

2.1.4 Planificación

En cualquier Sistema Operativo en el que existan múltiples actividades en ejecución concurrente, se hace imprescindible la existencia de un mecanismo que determine la política de planificación soportada.

2.2 Requerimientos no funcionales

Los requisitos no funcionales suponen criterios no operacionales que miden la bondad con que el sistema operativo lleva a cabo sus funciones.

2.2.1 Abstracción única y sencilla

Uno de los principales requerimientos no funcionales de cualquier sistema operativo es el principio de la **Economía de Conceptos**. Las abstracciones deben ser, por un lado, la menor cantidad posible, favoreciendo así la economía de conceptos y un mejor

rendimiento y, por otro lado, lo más sencillas posible de forma que resulten simples y comprensibles a los programadores.

2.2.2 Flexibilidad del entorno

A la hora de definir las abstracciones, es necesario tener en cuenta que, si bien estas generalizan el comportamiento del hardware facilitando su uso, también limitan la flexibilidad de manipular de forma concreta el hardware con un software determinado[Nut98].

Sin embargo, en general, diferentes aplicaciones tienen diferentes requisitos. De esta forma, hacer el sistema flexible, significa dotar al sistema de una implantación flexible que pueda cambiar según la variación de los requisitos de cada aplicación[WU95].

2.2.3 Eficiencia

Es evidente que, a cambio de contar con una plataforma más sencilla de utilizar, el usuario puede renunciar a cierto grado de eficiencia. Sin embargo, esto no es óbice para que el Sistema Operativo intente sacar el máximo partido al hardware subyacente.

3 Modelo de procesos tradicional

3.1 Abstracciones

Los sistemas de gestión de procesos más tradicionales, como el de Unix[Bac86], ofrecen la abstracción de **proceso** para introducir la concurrencia y representar así las actividades del sistema. Los procesos, gestionados enteramente por el sistema operativo, son la abstracción de la máquina subyacente que estos sistemas operativos ofrecen al usuario para la ejecución de sus programas. En ella se recogen los recursos necesarios para la ejecución de un programa de usuario, además de determinar el ámbito de acceso o marco de protección [TW98, Sta98, Bac92].

3.2 Concurrencia

La mayoría de los sistemas operativos modernos permite que muchos procesos estén activos a un tiempo en el sistema. Para estos procesos, el sistema proporciona las características de una máquina virtual[Nut97], dando a cada uno de ellos la ilusión de que es el único en la máquina. Así, los programadores escriben sus aplicaciones como si fuesen los únicos que se están ejecutando en el sistema.

Realmente, todos los procesos activos compiten por los recursos existentes en el sistema y el sistema operativo actúa como el gestor de recursos, distribuyéndolos de forma eficiente. Si la máquina dispone de un único procesador, sólo un proceso podrá ejecutarse realmente en un momento dado y el resto de los procesos deben suspender su ejecución esperando por la CPU o por cualquier otro recurso.

3.3 Comunicación entre procesos

La única abstracción en el sistema que tiene sentido que se comunique es el proceso, ya que todas las demás sólo existen bajo el control de un proceso. Los sistemas operativos ofrecen distintas posibilidades de comunicación como las tuberías (del inglés *pipes*), los mensajes o la memoria compartida[SG98].

Todos estos mecanismos son gestionados completamente por el sistema operativo que crea, gestiona y destruye las estructuras de datos internas necesarias para implementarlos, a la vez que define la semántica de funcionamiento de todos ellos. Para utilizarlos, el programa invoca al sistema solicitando un servicio de una forma tradicional (**llamada al sistema**).

3.4 Planificación de Procesos

Los mecanismos de planificación del procesador en los sistemas operativos tradicionales son bastante rígidos. Frecuentemente, dividen la CPU entre los procesos utilizando una política de planificación fija que, sin embargo, no es capaz de satisfacer las necesidades de planificación de muchos de los procesos en ejecución.

Dado que la implantación de la política concreta soportada por el sistema operativo está íntimamente ligada al núcleo del sistema operativo[FS96], la modificación de la misma por otra es bastante difícil.

3.5 Interfaz del sistema operativo

El sistema operativo ofrece una interfaz de programación a los procesos definidos por los usuarios. Esta interfaz viene definida por una serie de funciones denominadas **llamadas al sistema**. Las llamadas al sistema tienen, normalmente, la forma de una llamada a función aunque se ejecutan en modo supervisor en el contexto del proceso.

El conjunto de llamadas al sistema o interfaz que el sistema operativo ofrece a los procesos es, normalmente, un conjunto fijo y predefinido sin que se ofrezca ninguna posibilidad de interacción por parte del usuario.

3.6 Ventajas e Inconvenientes del modelo de procesos

3.6.1 Ventajas

Entre las principales ventajas que se pueden citar destacan, principalmente dos.

Modelo muy conocido y ampliamente extendido

En primer lugar, es un modelo muy conocido por los programadores, está ampliamente difundido y muy implantado con lo que no es necesario un trabajo previo de estudio o entrenamiento para producir en este modelo.

Modelo Robusto

Se trata además de un modelo robusto. El sistema operativo previene los ataques al sistema operativo o a otros procesos mediante la idea de procesos en espacios de direcciones propios y privados.

3.6.2 Inconvenientes

El modelo de procesos descrito anteriormente, aunque conocido y robusto, tiene dos importantes limitaciones[Vah96, MSL+91].

Falta de adecuación ante los nuevos paradigmas

En primer lugar, el desarrollo de un nuevo paradigma como fue el cliente/servidor dio lugar a la aparición de gran cantidad de aplicaciones en las que es necesario ejecutar varias tareas concurrentemente, aunque compartan recursos como el espacio de direcciones. Estos procesos son inherentemente paralelos y necesitan un modelo de programación que soporte este paralelismo.

Los sistemas operativos basados en el modelo de procesos, no ofrecen soporte explícito para este nuevo paradigma y dependen de costosos mecanismos para la gestión de múltiples operaciones.

Igualmente, el soporte del paradigma de OO ve cómo las abstracciones ofrecidas por el modelo de procesos tradicional no son capaces de dar un soporte adecuado sin un coste excesivo.

Rendimiento Pobre

En segundo lugar, los procesos tradicionales no aprovechan las ventajas inherentes a las nuevas arquitecturas multiprocesador, dado que un proceso sólo puede estar en un procesador a un tiempo. Por tanto, las aplicaciones deben crear varios procesos separados y repartirlas en los procesadores disponibles. Además, es necesario ampliar el sistema operativo ofreciendo modos de compartir memoria y recursos y de sincronizar las tareas.

La necesidad de usar múltiples procesos en las aplicaciones provoca una sobrecarga inherente a la creación de los procesos y a la necesidad de utilizar mecanismos IPC para la comunicación de los mismos debido a las barreras establecidas por los espacios de direcciones privados.

3.7 Integración de la OO con el Modelo de Procesos

La unificación de objeto y proceso es una aproximación muy sencilla a la integración de la computación en los sistemas orientados a objetos ya que objetos y procesos comparten algunas características comunes como encapsulación de datos privados protegidos de accesos externos o comunicación a través de algún modo de paso de mensajes. Existen dos posibilidades de integrar objetos y procesos[GKS94].

3.7.1 Un Proceso contiene un objeto.

Se trata de la aproximación de servidor tradicional en la que un objeto se identifica o representa con un proceso. Tiene la ventaja de que los objetos no sólo encapsulan su

estado privado y la descripción del comportamiento, sino también sus actividades, en este caso, su actividad. Facilita la consistencia del estado interno de los objetos al comportarse los objetos como un monitor.

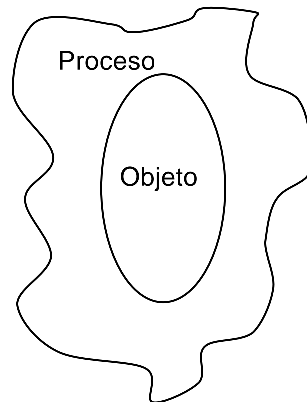


Figura 3.2 Proceso = Objeto

El mayor inconveniente de esta aproximación es la reducción del paralelismo que se basa únicamente en la existencia de varios procesos en distintos objetos. Además de tratarse de abstracciones demasiado pesadas si la granularidad del objeto es pequeña.

3.7.2 Un proceso encapsula múltiples objetos.

Se trata de la aproximación más tradicional, que se corresponde con la clásica separación de procesos y recursos, en este caso representados en los objetos, en los sistemas Operativos. Los objetos tienen sentido dentro de los procesos, como parte de su espacio de direcciones. Sería el caso de objetos C++ encapsulados dentro de procesos UNIX. En este caso, dotar al objeto de entidad propia exige costosos mecanismos para poder migrar los objetos entre espacios de direcciones.

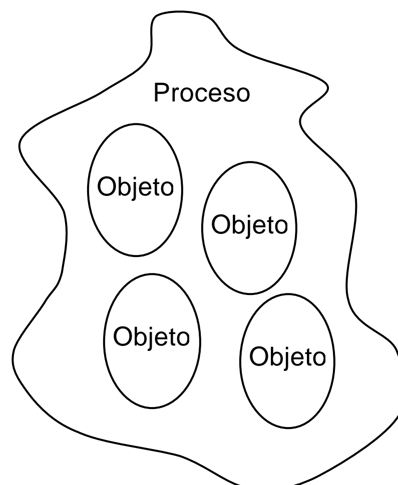


Figura 3.3 Proceso = {Objetos}

4 Modelo de Tareas e Hilos

En este modelo, del que se puede citar como máximo exponente al sistema operativo Mach[ABG+86, Ras86], se presenta un entorno de ejecución más avanzado que permite un mejor rendimiento.

En lugar de ofrecer una única abstracción monolítica, los sistemas operativos basados en esta abstracción, descomponen esta visión en dos abstracciones fundamentales[JR86, Vah96].

1. **Tarea o Task**. También llamado en algunas obras proceso por analogía con su antecesor, se trata de una entidad pasiva que representa el entorno de ejecución en que los hilos pueden ejecutarse. Para ello, engloba a un espacio de direcciones virtual y acceso protegido al conjunto de recursos del sistema como procesadores, puertos, etc.
2. **Hilos, Threads** o **Procesos ligeros**. Son la unidad básica de utilización de la CPU y se ejecutan dentro del contexto de una tarea compartiendo el espacio de direcciones y otros recursos, lo que obliga a sincronizar el acceso a aquellos datos que puedan ser accedidos concurrentemente por más de un hilo, ya que no hay nada que prevenga los conflictos que se pueden presentar. Sin embargo, cada uno de ellos tiene su propia pila y contexto de registros además de poder mantener algunos datos de estado privados.

1.1 Comunicación entre hilos

La comunicación entre hilos, la gestiona normalmente el propio sistema operativo y se lleva a cabo mediante el **envío de mensajes** entre hilos. El envío y recepción de mensajes se lleva a cabo mediante primitivas de envío y recepción de mensajes explícitas, y cualquier hilo dentro de una tarea puede enviar o recibir mensajes.

1.2 Posibles Modelos de Hilos

1.2.1 Procesos ligeros

Un **proceso ligero** o LWP, del inglés *Lightweight Process*, es un hilo soportado por el núcleo. Cada proceso o *task* puede tener uno o más y, cada uno de ellos puede realizar llamadas al sistema y bloquearse si estas son bloqueantes.

Cada hilo se planifica y ejecuta independientemente, lo cual, en un sistema multiprocesador, facilita que los procesos puedan disfrutar de los beneficios del paralelismo real dado que cada uno puede ser despachado para ejecutarse en un procesador distinto.

Ventajas e Inconvenientes de los procesos ligeros

Este modelo presenta múltiples **ventajas** sobre el modelo de procesos presentado como son una mayor adecuación al procesamiento paralelo(múltiples hilos cooperando en el mismo problema) y un mejor rendimiento en ambientes multiprocesadores.

Entre los **inconvenientes** cabe destacar [Vah96, MSL+91, ABL+91].

- **Demasiado pesados**. La mayoría de las operaciones como creación, sincronización, etc., requieren llamadas al sistema, caras debido al cruce de fronteras de protección, lo que los hace inadecuados para aplicaciones que utilizan una gran cantidad de hilos o que los crean y destruyen frecuentemente.
- **Costosos**. Cada hilo consume una cantidad significativa de recursos del sistema operativo, incluyendo memoria física para la pila kernel, etc. Debido a esto, un sistema no puede soportar un número excesivamente grande de hilos.
- **Generales**. Dado que un sistema tiene implementado un único modelo de hilos, este debe ser suficientemente general para soportar cualquier aplicación que desee utilizarlos, lo que implica que los hilos tendrán una gran sobrecarga que, la mayoría de las veces, no se utilizará.

1.2.2 Hilos de Usuario

Es posible proporcionar completamente la abstracción de hilo a nivel usuario lo que se consigue a través de librerías de hilos como los C-Threads de Mach [CD88] o los pthreads de POSIX[IEE94].

Estas librerías proporcionan todas las funciones para crear, sincronizar, planificar y gestionar hilos y no necesitan ninguna ayuda especial por parte del kernel, que no tiene ninguna información acerca de ellos, ni cuántos son, ni su estado, ni tampoco se requiere la intervención del mismo para la interacción de los hilos o para el cambio de hilo, lo que hace que sea más rápida.

Pueden implementarse dentro de procesos convencionales y la gestión de los mismos se hace por medio de un sistema en tiempo de ejecución que se añade y las estructuras de datos para la planificación se mantienen en memoria compartida.

Ventajas e Inconvenientes de los hilos a nivel usuario

Los hilos de usuario tienen gran cantidad de **beneficios**[Vah96].

- Proporcionan un **modo natural de programación** y un paradigma de programación síncrona que esconde la complejidad de las operaciones asíncronas en la librería de hilos.
- **Rendimiento**. Consumen pocos recursos kernel y el rendimiento mejora al implementar la funcionalidad a nivel usuario, sin usar llamadas al sistema, lo que evita la sobrecarga de procesar la interrupción software y el cruce de fronteras de protección.

Sin embargo, los hilos de usuario presentan también varias **limitaciones** debidas, principalmente, a la separación de hilo núcleo y hilo usuario[Vah96, ABL+91].

- **Falta de Protección**. Dado que el núcleo no conoce la existencia de hilos de usuario, no puede usar los mecanismos de protección para proteger a unos de otros.
- **Planificación a dos niveles**. El planificador de la librería de hilos planifica hilos de usuario y el del núcleo planifica LWP y no existe comunicación entre ellos.
- **Sincronización inadecuada**. Las llamadas al sistema deben ser no-bloqueantes, ya que eso implicaría que el LWP que contiene al hilo que realiza la llamada podría bloquearse, aunque tuviera otros hilos listos.

- **Peor rendimiento en ambientes multiprocesadores**. En ambientes multiprocesadores, el modelo de hilos de usuario compartiendo LWP no aprovecha el hardware subyacente.

1.2.3 Scheduler Activations

[ABL+91] describe una nueva arquitectura de hilos que combina las ventajas de los LWP con los hilos de usuario.

El principio básico es la integración entre los hilos de usuario y el núcleo, cada uno de los cuales lleva a cabo una parte del trabajo.

El núcleo tiene control absoluto acerca de la asignación de procesadores entre espacios de direcciones, incluyendo la posibilidad de cambiar el número de procesadores asignados a una aplicación.

El sistema de hilos a nivel usuario sabe cuántos y qué procesadores tiene asignados y tiene un control absoluto sobre la planificación, es decir, decide cuáles de sus hilos se ejecutan en esos procesadores. Puede solicitar procesadores adicionales o liberar alguno de los que tenga asignados. Si un hilo se bloquea en una operación del núcleo, el proceso no pierde el procesador, sino que el núcleo informa a la librería que, inmediatamente planifica algún otro hilo de usuario en ese procesador.

Así, el núcleo notifica al planificador de hilos a nivel usuario de cada evento que le afecte, en lugar de interpretar por sí mismo esos eventos, como en los sistemas tradicionales.

El sistema de hilos a nivel usuario notifica al núcleo todos aquellos eventos que puedan afectar a decisiones de asignación de procesadores.

Upcalls y Scheduler Activations

El núcleo utiliza dos nuevas abstracciones para llevar a cabo tales ideas: upcall y scheduler activations.

Una **upcall** es una llamada hecha desde el núcleo a la librería de hilos. La **scheduler activation** es un contexto de ejecución que puede usarse para ejecutar un hilo. Es similar a un LWP y tiene su propia pila.

Cuando el núcleo hace una upcall, pasa una scheduler activation a la librería. El planificador de hilos del espacio de direcciones utiliza este contexto para gestionar el evento modificando las estructuras de datos de los hilos a nivel usuario, ejecutando hilos a nivel usuario y realizando llamadas al sistema.

En cada instante el núcleo asigna a cada proceso una activation por cada procesador que tenga asignado.

Ventajas

- **Rapidez** Son muy eficientes dado que la mayor parte de las operaciones no involucran al kernel.
- **Mayor integración nivel núcleo/ usuario**. El kernel informa a la librería de los eventos de requisamiento y bloqueo, así, la librería puede tomar decisiones de planificación y sincronización más adecuadas.

1.3 Integración de la OO con el Modelo de Tareas/Hilos

La integración de objeto en el modelo descrito es similar a la de procesos. Frente al modelo anterior, aumenta el nivel de concurrencia al haber múltiples hilos dentro de una tarea.

Las tareas engloban todos los recursos necesarios y, entre estos recursos, estarían los objetos. Dentro de cada tarea habría múltiples objetos que estarían compartidos por los distintos hilos encapsulados en dicha tarea. De esta forma, la invocación de un método supondría encontrar el objeto referenciado y, mediante algún mecanismo de carga y enlace dinámicos, enlazarlo al espacio de direcciones de la tarea. Un ejemplo de sistema de este tipo es el entorno Guide[BBD+91, BCC+91] que utiliza el micronúcleo de Mach para construir, basándose en las abstracciones que este proporciona, un sistema de soporte de objetos.

2 Problemas de los mecanismos de computación tradicionales

Los modelos vistos, suponen mantener en el Sistema Operativo el soporte a una abstracción tradicional tipo el proceso o el hilo. Sin embargo, dado que es necesario dar soporte al objeto en el sistema operativo, se hace necesaria la integración de ambas o, al menos, ofrecer mecanismos que las conecten y establezcan relaciones entre ambas.

Pero, como se verá a continuación, ambas entidades no acaban de encajar del todo bien, provocando una serie de problemas de difícil solución, si se mantiene un esquema tradicional. De todos esos problemas, se señalan aquí, únicamente, los más destacados.

2.1 Pobre adecuación para el soporte de Objetos

Los sistemas Operativos tradicionales no son una buena base para construir sobre ellos un sistema Operativo OO[LJP93].

Son dos las razones fundamentales que justifican esta afirmación. En primer lugar, debido a lo poco adecuadas que resultan las abstracciones que ofrecen y la consiguiente desadaptación de impedancias entre el soporte y la abstracción de objeto que se desea soportar.

En muchos casos, estas abstracciones resultan además, insuficientes. Así, por ejemplo, los lenguajes OO necesitan soporte para entidades de cualquier granularidad, incluidos objetos de grano fino. Sin embargo, la mayoría de los sistemas operativos proporcionan una abstracción como el proceso o la tarea como la entidad más pequeña que soportan y queda en manos de los compiladores establecer la correspondencia del modelo de grano fino del lenguaje con el modelo de grano grueso del sistema.

De la misma forma, en algunos casos, se necesita soporte para objetos activos, con capacidad de procesamiento, como los actores [Agh86] y los sistemas operativos sólo ofrecen entidades pesadas como los procesos. Por ejemplo, establecer una correspondencia entre actividades de grano fino y procesos Unix muy pesados (considérese por ejemplo, el caso de los objetos Emerald[JLH+88]) es un ejemplo obvio de la desadaptación entre los conceptos del sistema operativo y del lenguaje.

2.2 Demasiadas Abstracciones

En el modelo de integrar los conceptos tradicionales al soporte de OO, se añaden, además de las tradicionales, todas aquellas abstracciones necesarias para dar soporte al concepto de objeto.

La aparición de todas estas abstracciones se debe a la aparición de problemas de flexibilidad o integración de las abstracciones anteriores a ellas y que ellas intentaban solucionar. Aparecen así gran cantidad de abstracciones como tarea, hilo, espacio de direcciones, procesos, etc. Esto elimina la simplicidad y la economía de conceptos.

2.3 Falta de Uniformidad

Los mecanismos de gestión adolecen de falta de uniformidad, a la hora de integrarse con características avanzadas como la persistencia o la distribución.

Así, la invocación de métodos en objetos que se encuentran en el espacio de direcciones del proceso o la tarea se implanta de forma directa mediante llamada a procedimiento. Pero, para invocar un método en un objeto remoto o en almacenamiento secundario, es necesario un mecanismo diferente, mediante el cual el proceso o tarea se difunden o migran a otros nodos.

2.4 Inflexibilidad

En los sistemas operativos tradicionales, el sistema de computación ofrecía a las aplicaciones abstracciones tipo proceso o hilo que representaban la computación de la máquina, a la vez que proporcionaban interfaces para la planificación, la sincronización y la concurrencia.

Estos mecanismos de gestión se proporcionan de forma fija y estática y, sin embargo, en muchos casos, las aplicaciones requieren niveles de funcionalidad y rendimiento que los sistemas operativos son incapaces de proporcionar.

El ejemplo más ilustrativo es probablemente el de la planificación. Los mecanismos de planificación de los sistemas operativos formn parte componente del sistema operativo.

Los sistemas operativos tradicionales controlan el reparto de los recursos (entre ellos el tiempo de CPU o de ejecución) utilizando un esquema de planificación fijo, normalmente, aquellos procesos que tienen igual prioridad se planifican en modo round-robin y cada uno de ellos disfruta de un quantum de tiempo. Si aparece algún proceso más prioritario puede arrebatar la CPU al proceso actual aunque este último aún no haya terminado de usar su tiempo.

Frecuentemente, por no decir siempre, han soportado únicamente una única política de planificación y además, esa política era fija e inamovible. Algunas veces se proporcionan variantes de esta política básica, como varias clases de planificación a las que se asignan hilos con diferentes propósitos. Sin embargo, incluso estas variantes suelen estar integradas en la implementación del sistema y no pueden ser adaptadas fácilmente a las necesidades de las aplicaciones individuales.

Esto provoca un grave problema de integración con las aplicaciones actuales que, en la mayoría de los casos son aplicaciones multihilo y, en muchos casos necesitan ser

capaces de definir una estrategia particular para sus hilos (servidores de ficheros, servidores de red, ...).

3 Modelo Objeto/Hilo

Debido a la gran popularidad y aceptación alcanzadas por el paradigma de la OO, se han diseñado e implementado varios lenguajes de programación concurrentes OO, basándose en el modelo de objetos concurrentes donde cada objeto es una entidad activa[YT87a].

Sin embargo, desde el punto de vista del sistema operativo, cada objeto era un proceso con un único hilo de control, lo que hace necesario escribir gran cantidad de código adicional para soportar las abstracciones necesarias para los objetos.

Una consecuencia de la aplicación directa de los modelos de computación anteriores como intento de introducir la concurrencia en los Sistemas orientados a objetos, fue la aparición del modelo objeto/hilo[MHM+95].

El modelo objeto-hilo pretende incorporar las construcciones de concurrencia en los sistemas convencionales orientados a objeto y para hacerlo, introduce **dos entidades**, el **objeto** y el **hilo** ambos con la semántica tradicional.

3.1 Objetos e Hilos

Objetos en el modelo Objeto/Hilo

En este modelo, los objetos son espacios de direcciones con nombres que proporcionan almacenamiento de datos y métodos para la manipulación. Son entidades pasivas que ofrecen las funciones de compartir datos y sincronización.

Hilos en el modelo Objeto/Hilo

Por su parte, los hilos representan el flujo de control en el sistema por medio de la invocación y posterior ejecución de métodos en los objetos. Normalmente, no están ligados a un espacio de direcciones u objeto, sino que cada hilo puede ejecutar código del método de un objeto en un momento determinado y puede atravesar los objetos migrando a otro a medida que invoca sus métodos.

3.2 Ventajas e Inconvenientes del Modelo

Ventajas de este modelo

- Es un modelo que proporciona un **buen rendimiento**, dado que existen múltiples hilos en ejecución en un instante de tiempo.
- Se trata de un **modelo familiar** porque, en la mayoría de los casos, se utiliza como lenguaje base un lenguaje de programación tradicional y, en cualquier caso, está bastante extendido en librerías de hilos con lenguajes tradicionales de programación.

Inconvenientes

- **Nueva entidad**. Dado que se trata de introducir la concurrencia en un lenguaje que no tenía tal habilidad y, por tanto, tampoco sabía cómo controlar la ejecución concurrente, es necesario introducir una nueva entidad externa y no programable que representa el hilo.
- **Exclusión Mutua**. Dado que los hilos se ejecutan independientemente, es sensible a los problemas de exclusión mutua, por ejemplo, cuando dos o más hilos ejecutan simultáneamente el mismo método. La aparición de código externo al objeto para realizar la sincronización del propio objeto añade complejidad a la programación, especialmente si se usa combinado con el mecanismo de la herencia. Ver Figura 3.4.

```
// Definición ClaseA
Class ClaseA {
Private:
  Int x;
Public:
  Void método1() {x=0;}
// Este método deberá ser modificado cuando se defina la ClaseB que hereda de
ClaseA → Problemas con la herencia
}
// Definición ClaseB, posterior a la definición de ClaseA y que hereda de esta
Class ClaseB: public ClaseA {
Public:
  Void método2()
{
lock(x);
x=10;
// conflicto con ClaseA::método1
unlock(x);
}
}
```

Figura 3.4 Ejemplo que muestra la poca adecuación del modelo objeto hilo

Como representantes de este modelo se puede citar al Sistema Operativo Clouds[DLA+91].

Capítulo IV

Visión general de algunos Sistemas Operativos Relevantes

A tenor de lo visto en anteriores capítulos, fundamentalmente en el capítulo II, se revisarán aquí algunos de los sistemas operativos más relevantes, haciendo especial hincapié en la organización interna o arquitectura interna aplicada a los mismos.

La elección de los sistemas que aquí se describen se ha hecho de forma flexible. No se trata de una descripción de Sistemas Operativos de una determinada clase, de hecho, algunos de los sistemas presentados aquí, no pueden ser considerados sistemas operativos, sino máquinas virtuales o entornos de programación.

Sin embargo, se han estudiado con el fin de abarcar el espectro más amplio posible y poder así capturar la máxima cantidad posible de características interesantes que no tienen por qué venir únicamente del campo de los sistemas operativos.

De entre todos los que aquí se presentan, se observará que muchos de ellos presentan características de OO que ya en el Capítulo II se presentaba como una aproximación a estudiar.

Con ello se intenta encontrar aquellas características de los mismos, que resulten relevantes para esta tesis, apoyando los principios que en ella presentamos.

1 Choices: Un sistema operativo Orientado a objetos basado en la reusabilidad

Choices[CIM+93] es una familia de sistemas operativos para sistemas multiprocesador de memoria distribuida y compartida.

1.1 Arquitectura del Sistema

El diseño de Choices se captura mediante un **marco de aplicación** (*framework*) que describe los componentes del sistema en abstracto y la manera en que interactúan. El marco de aplicación es una jerarquía de clases C++.

El **núcleo** en tiempo de ejecución es un núcleo monolítico, implementado como un conjunto de objetos C++ que resultan de una instanciación concreta del marco. Estos objetos implementan los diferentes aspectos del sistema operativo, como la interfaz con las aplicaciones (espacio del usuario) y con el hardware, y los recursos, políticas y

mecanismos del sistema. En cualquier caso, son los objetos propios de un programa en C++.

1.2 Abstracciones de Choices

A pesar de estar implementado de manera interna con un lenguaje orientado a objetos, las abstracciones que proporciona Choices son las de un sistema operativo más convencional: **procesos** (hilos), **dominios** (espacios de direcciones virtuales) **y objetos de memoria**. Existe una clara división entre el espacio del sistema y el de usuario. Siempre existe un dominio del sistema en el que hay procesos del sistema que funcionan en modo supervisor. En modo usuario puede haber varios dominios de usuario con procesos de aplicación.

Sin embargo, la interfaz del sistema operativo (interfaz a los servicios del núcleo) se proporciona a las aplicaciones mediante un conjunto de objetos colocados en el núcleo. Las aplicaciones acceden a los servicios del sistema invocando métodos de estos objetos. La manera de cruzar la frontera entre el usuario y el sistema es mediante un **objeto representante** (*proxy*) del objeto del sistema dentro del espacio del usuario. Combinando estos representantes con listas de control de accesos y servidores de nombres se establecen los mecanismos de seguridad del sistema.

1.3 Computación en Choices

Choices presenta un modelo de computación tradicional basado en procesos(hilos), que representan la unidad básica de ejecución, aunque se construye utilizando TOO, lo que hace que el proceso se encapsule dentro de un objeto.

Sin embargo, su comportamiento es el de un proceso tradicional.

Para el sistema de gestión de procesos y planificación, Choices implementa 4 clases abstractas fundamentales.

- **Process**. Representa un proceso y su contexto.
- **Processor**. Representa un procesador físico.
- **ProcessContainer**. Representa un almacén (del término inglés, *repository*) de procesos.
- **Exception**. Encapsula los gestores de interrupciones software y hardware.

Representación de la abstracción de proceso

Cada proceso se representa por una instancia de la clase **Process** que abstrae un proceso tradicional, por lo que los datos que compone el estado de un objeto de esta clase son similares a los que representa el contexto de un proceso: **Memoria Virtual**, representada por el objeto *Domains*, **Estado** y **Contexto de los registros**.

Los métodos de estos objetos *Process* alteran su memoria virtual, manipulan sus parámetros de planificación y pueden gestionar el requisamiento y el cambio de procesos.

Contenedores de Procesos

La evolución de los procesos en Choices se basa en la existencia de objetos **Contenedores de Procesos**, instancias de la clase **ProcessContainer** o de una de sus subclases. Estos objetos contenedores, como su nombre indica, contienen objetos procesos que se mueven de uno a otro contenedor, en función de su evolución en tiempo de ejecución.

Los contenedores de procesos son una pieza fundamental del mecanismo de computación de Choices dado que las primitivas de planificación, bloqueo y despacho de procesos se construyen usando instancias de la clase *ProcessContainer* y de sus subclases y los algoritmos de planificación y despacho de procesos en Choices involucran transferencias de procesos entre Contenedores de Procesos.

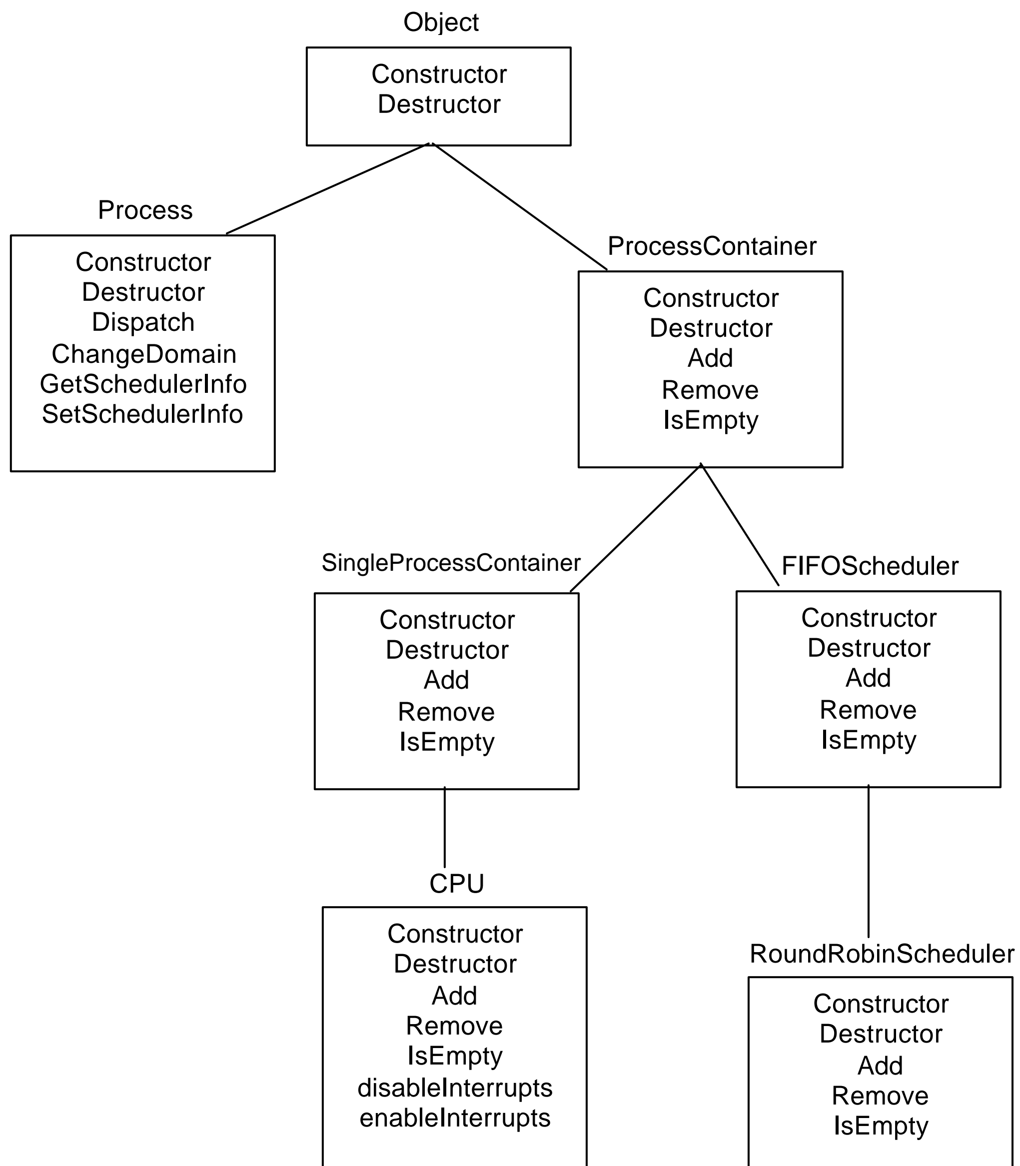


Figura 4.1 Jerarquía básica de Clases para la Gestión de Procesos

Las subclases de la clase *ProcessContainers* imponen disciplinas de colas a los procesos que contienen y definen los métodos **add**, **remove** y **isEmpty**.

1.4 Planificación

La planificación en Choices sigue también una aproximación tradicional, en cuanto a que son los planificadores los que se encarga de determinar el siguiente proceso a ejecutar.

Jerarquía de Planificadores

Los planificadores en Choices son objetos instancia de alguna clase de la jerarquía de planificadores. Esta jerarquía es extensible, permitiendo añadir nuevos planificadores a la misma que se adapten mejor a determinados requerimientos por parte de las aplicaciones.

Objetos involucrados en la Planificación

Los planificadores son subclases de la clase ProcessContainer, es decir, son contenedores de procesos y es a esos procesos a los que se le aplica la política de planificación definida por el planificador concreto de que se trate.

Otros planificadores que existen simultáneamente en el sistema en tiempo de ejecución, pueden aplicar otras políticas diferentes a otros procesos.

Además de los planificadores y los procesos, en la planificación también se ven involucrados los objetos **CPU**, objetos que, como su nombre indica, representa al procesador físico subyacente.

Se trata también de un Contenedor de procesos que, lógicamente, sólo contendrá un proceso simultáneamente, el proceso en ejecución en cada instante dado.

Cada CPU tiene un planificador asociado que se encargará de proporcionarle procesos para ejecutar cuando esté ocioso.

El proceso de Planificación

La planificación en Choices se basa en el movimiento de procesos entre contenedores. El planificador interacciona con los objetos procesos y CPU, moviendo al contenedor CPU un objeto proceso cuando este esté ocioso.

Para elegir el proceso a mover a la CPU, el planificador aplica su política de planificación a los objetos que contiene. Para ello, cada proceso tiene asociada una instancia de la clase **SchedulerInformation** que es mantenida por el planificador.

1.5 Manejo de Excepciones

Choices modela las excepciones como una **jerarquía de clases**. La raíz de esta jerarquía es la clase abstracta **Exception** que define el método **raise** para interceptar la condición de excepción.

Las excepciones son los objetos más básicos de Choices cuyos métodos causan movimientos de procesos entre Contenedores de Procesos.

Cuando se produce una interrupción hardware o software, el mecanismo de gestión de interrupciones invoca código específico de la arquitectura subyacente, que se encarga de salvar parte del contexto del proceso en ejecución y enviar el mensaje *handle* al objeto excepción correspondiente.

1.6 Semáforos

La clase **Semaphore** presenta los métodos P y V y se le asocia una cola de procesos suspendidos que serán transferidos desde la CPU a esta cola cuando el proceso requiera una operación P bloqueante en el semáforo.

De la misma forma, estos procesos bloqueados se mueven desde esta cola a la cola de listos del sistema cuando otro proceso ejecute el método V del semáforo.

La operación P del semáforo utiliza el método *raise* de la clase **SemaphoreException** que mueve el proceso del contenedor CPU a la cola de espera del semáforo y elige otro proceso para ejecutarse de entre los que están en la cola de listos del sistema.

1.7 Jerarquía de marcos: Reutilización del Código

La importancia de Choices está en haber sido pionero en la utilización de la orientación a objetos mediante marcos de aplicación a la construcción de una familia de sistemas operativos.

Figura 4.2 Marco de aplicación superior del sistema Choices.

La estructura es la de una jerarquía de marcos. El marco superior es un conjunto de clases abstractas que describe los componentes básicos del sistema y restricciones que deben cumplir los sub-marcos.

Los sub-marcos representan subsistemas del sistema operativo que introducen especializaciones (aún abstractas) de las clases abstractas del marco anterior y nuevas restricciones.

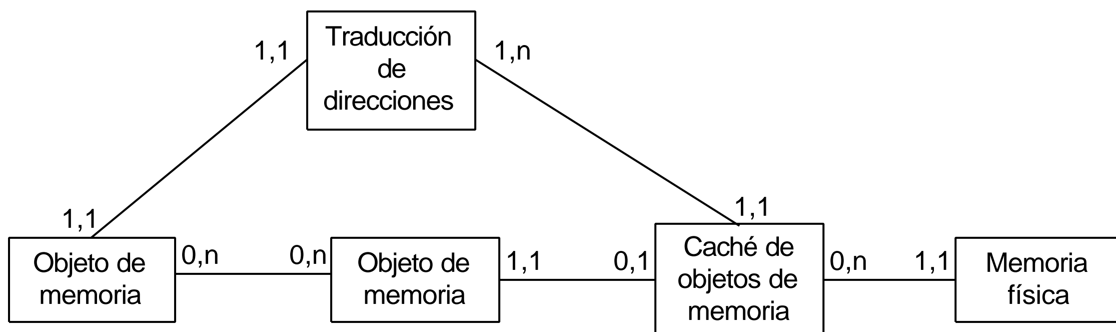


Figura 4.3 Sub-marco de aplicación para la gestión de memoria de Choices.

Un sistema operativo concreto, es decir una instanciación concreta del marco de aplicación Choices, se construye sustituyendo las clases abstractas del marco por clases concretas que definen una implementación determinada.

1.8 Crítica

A pesar de utilizar la OO, se utiliza de una manera estática y restringida, y para dar soporte a abstracciones de sistemas operativos convencionales.

1.8.1 Separación usuario / sistema: sólo flexibilidad estática

Se separan claramente los elementos del usuario y los elementos fijos del sistema. Esta separación impide aplicar la flexibilidad dinámica a los elementos del sistema. Sólo se consigue la flexibilidad estática al estar desarrollada la implementación como un marco de aplicación OO.

1.8.2 Uso restringido al C++

La utilización del sistema y del soporte de objetos está restringido únicamente al lenguaje de construcción del mismo, que es C++. El modelo de objetos del C++ no es muy adecuado para el sistema integral, puesto que en tiempo de ejecución un objeto C++ es simplemente una zona de memoria. Se pierde casi toda la semántica del modelo de objetos.

1.8.3 Falta de uniformidad en la Orientación a Objetos

A pesar de estar implementado mediante objetos, las abstracciones que utiliza el sistema son las abstracciones tradicionales de proceso, espacio de direcciones, fichero, etc.

1.8.4 Mecanismo de computación tradicional no Orientado a Objetos: Abundancia de Abstracciones y falta de integración con los objetos

Choices no introduce un mecanismo de computación realmente orientado a objetos, sino que modela, aplicando técnicas de Orientación a Objetos, el mecanismo de computación tradicional basado en procesos, planificación de procesos y semáforos que los sincronizan. Se podría concluir que el mecanismo de computación de Choices es el mismo que el de Unix, pero escrito en C++.

Esto provoca la aparición de abundantes abstracciones tradicionales aunque representadas como objetos.

La falta de integración entre los objetos y la computación provoca una ausencia de uniformidad, dándose el caso de que, desde el punto de vista del usuario, los objetos se comunican invocando métodos, sin embargo, los procesos compartiendo objetos.

1.8.5 Código de Sincronización en los clientes

Choices es un sistema tradicional y, por tanto, ofrece mecanismos tradicionales para sincronizar sus procesos.

Así, el código de sincronización está exclusivamente dentro de los métodos de los objetos por lo que los semáforos son la herramienta más adecuada.

Sin embargo, esto presenta problemas de integración con otros mecanismos como la herencia.

1.9 Características interesantes

Como precursor del sistema Tigger, utiliza una jerarquía de clases como base de la implementación del sistema operativo, y además proporciona una interfaz de usuario orientada a objetos.

1.9.1 Jerarquía de clases de implementación y con interfaz OO para el usuario

Además de utilizar una jerarquía de clases en la implementación, que da flexibilidad estática al sistema, también utiliza una interfaz OO para acceder a los servicios del sistema. El usuario accede a objetos que tienen funcionalidad del sistema. Estos se organizan también mediante una jerarquía de clases que el usuario puede utilizar. En este aspecto si hay una uniformidad de uso OO. En el sistema integral es interesante que la funcionalidad del sistema operativo se proporcione mediante objetos que el usuario pueda usar como cualquier otro objeto, por tanto también estarán organizados en una jerarquía normal de clases.

2 Clouds: Sistema Operativo Distribuido

Clouds[DLA+91] es un sistema distribuido basado en los conceptos de la programación Orientada a Objetos. Integra un conjunto de nodos en un sistema conceptualmente centralizado compuesto de servidores de computación, servidores de datos y estaciones de trabajo del usuario.

2.1 Arquitectura del Sistema

Clouds es un sistema distribuido que integra un conjunto de máquinas en un entorno compacto que se comporta como una gran máquina. La implantación de Clouds se lleva a cabo por niveles, cada uno de los cuales ofrece abstracciones al nivel superior y se apoya en el inferior para construir las suyas.

2.1.1 El kernel RA

RA es el microkernel nativo que soporta los mecanismos básicos de manejo de memoria virtual y planificación a bajo nivel, proporciona mecanismos para manejo de memoria y procesador y exporta clases que heredarán los objetos del sistema.

Abstracciones Básicas

Implementa, fundamentalmente cuatro abstracciones: Segmentos, Espacios Virtuales, Isibas y Particiones.

Los **Segmentos** son secuencias de bytes sin significado, de longitud variable, persistentes y con un nombre único a nivel sistema que se denomina **sysname**.

Los **espacios virtuales** son rangos de direcciones virtuales y los **isibas** son las abstracciones de actividad.

Por último, las **Particiones** proporciona almacenamiento no volátil para segmentos.

2.1.2 Objetos del sistema

Son módulos de código compilados independientemente que tienen acceso a ciertas operaciones definidas por RA y que proporcionan los servicios del sistema operativo.

Conceptualmente, estos objetos son similares a objetos Clouds que viven en su propio espacio virtual, soportan invocaciones externas y tienen acceso a operaciones en el kernel de RA.

Son, entre otros, el manejador de hilos, manejador de objetos de usuario y red y ratp.

2.1.3 Objetos del usuario

Son objetos Clouds definidos por el usuario que proporcionan los servicios no críticos.

2.2 Computación en Clouds

2.2.1 Modelo Objeto/Hilo

Clouds presenta un modelo objeto/hilo, basado en estas dos abstracciones, el objeto y el hilo.

Objetos en Clouds

Un objeto en Clouds es un espacio de direcciones virtuales accesible únicamente por el código de los métodos. Son persistentes, de grano grueso y tienen un nombre global en el sistema, sysname.

En contra de lo que sucede en otros sistemas operativos orientados a objeto, un objeto Clouds no contiene ningún proceso, son objetos pasivos.

Un objeto en Clouds contiene código definido por el usuario, datos persistentes, un montón volátil y un montón persistente.

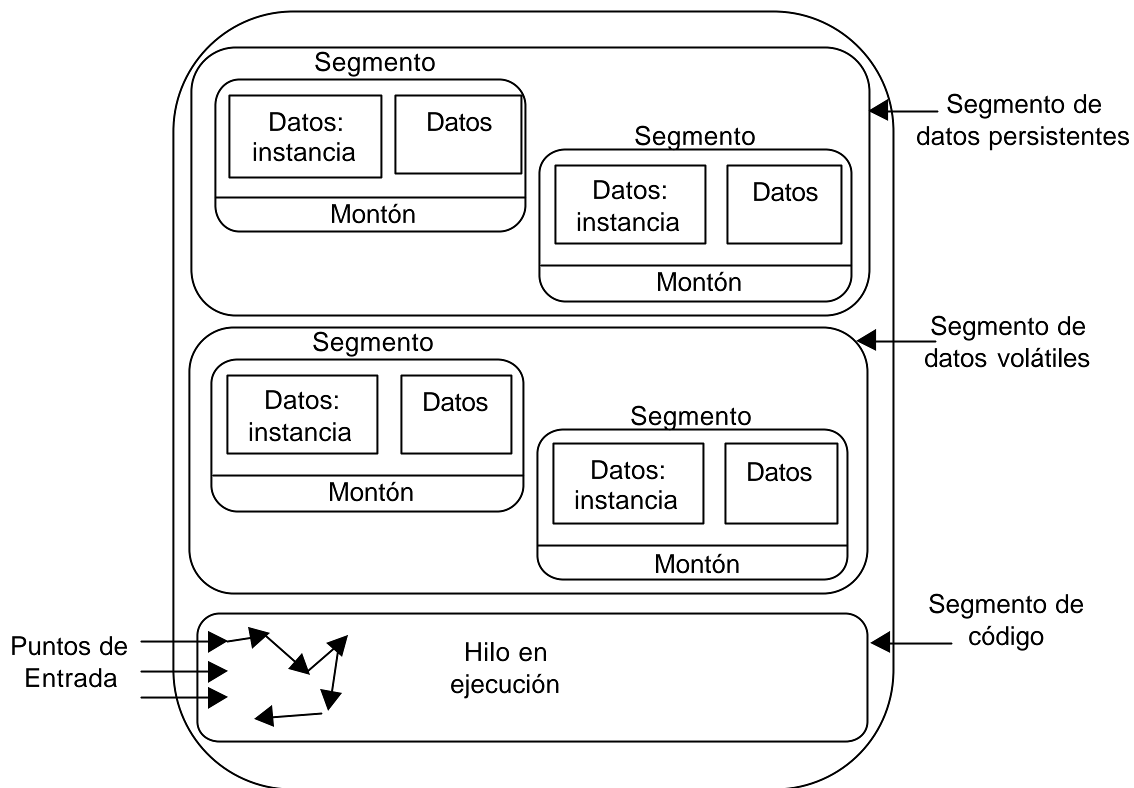


Figura 4.4 Objetos en Clouds.

Cada objeto tiene un identificador global único denominado **sysname**.

Hilos en Clouds

Se trata de la única forma de actividad en Clouds. Un hilo es un flujo de ejecución que ejecuta código en los objetos, atravesando los objetos que ejecuta. No está ligado a un espacio de direcciones virtuales, en contra de lo que ocurre con los procesos tradicionales. La transferencia de control entre espacios de direcciones es a través de la invocación de un objeto y la transferencia de datos entre espacios de direcciones siempre es a través del paso de mensajes.

Puede haber más de un hilo ejecutándose concurrentemente en un objeto. En este caso, estos hilos comparten el contenido del espacio de direcciones del objeto. El control de la concurrencia lo maneja el programador de los objetos mediante primitivas clásicas de sincronización soportadas por el sistema como semáforos y cerraduras.

2.2.2 Control de la Concurrencia y Consistencia del estado

Además de ofrecer mecanismos clásicos como los semáforos, Clouds permite a los programadores especificar un amplio rango de semántica de consistencia. Se basa en dos clases de hilos de ejecución, los **s-threads** o **standard threads** y los **cp-threads** o **consistency-preserving threads**.

2.3 Crítica

2.3.1 Falta de adaptación

El sistema operativo no se construye como Choices, como un framework para que sea personalizable, lo que limita la acción del usuario a los objetos de usuario, no permitiéndosele proporcionar objetos del sistema que proporcionasen un servicio del sistema de alguna forma mejorada o más adecuada al entorno actual.

2.3.2 Modelo Objeto/Hilo

El modelo objeto/thread presente en Clouds no establece ninguna relación estable entre un objeto y los hilos que se están ejecutando en él, lo que presentará problemas a la hora de migrar objetos con su computación a otras máquinas.

Clouds, además de presentar un modelo de objetos pasivo basado en la existencia de objetos hilo manejados por un gestor de hilos, diferente del manejador de objetos de usuario, basa sus mecanismos de sincronización en primitivas de sincronización como los semáforos y locks que pueden ser ofrecidos por el sistema.

2.4 Cuestiones Interesantes

2.4.1 Uniformidad Objetos Usuario/Sistema

Conceptualmente, no hay diferencia entre los objetos de usuario y los del sistema. Aunque a la hora de la implantación sí existe esa diferencia, esta no es visible al exterior, todos ellos son objetos Clouds, lo que implica que todos tienen la misma composición (segmentos, datos persistentes, datos volátiles, identificador único a nivel de sistema) y comportamiento.

2.4.2 Ofrece la interfaz OO

La forma de solicitar servicios del sistema operativo es idéntica a la forma en que se solicitan servicios de cualquier objeto de usuario, mediante la invocación de un método en un objeto.

2.4.3 Soporte para el control de la concurrencia

Es de los pocos sistemas que, además de proporcionar las primitivas de sincronización típicas, ofrece otros mecanismos de control de la concurrencia, como son los s-threads o los cp-threads que permiten establecer un grano más grueso de control, liberando al programador de tareas dificultosas de control de la concurrencia con el uso de semáforos.

2.4.4 No está restringido a ningún lenguaje concreto

No ofrece un soporte específico para un lenguaje de alto nivel concreto, sino que ofrece un soporte genérico sobre el que se podrán construir capas de soporte para distintos lenguajes de programación.

3 SPIN: Microkernel Extensible para Sistemas Operativos Específicos para la aplicación

El sistema operativo SPIN [BSP+95], es un micronúcleo que ofrece un conjunto de mecanismos y permite a las aplicaciones tomar decisiones acerca de políticas concretas descargando código que extiende el microkernel de forma segura.

Al existir una clara separación entre el espacio del usuario y el del sistema, esto además permite un aumento del rendimiento al no tener que realizarse cambios de contexto entre los espacios.

Sin embargo la adición de código de usuario al sistema dentro del espacio del núcleo presenta un problema de seguridad. Este código de usuario al estar dentro del núcleo podría, bien por errores de programación o maliciosamente, corromper estructuras y/o código del sistema operativo.

3.1 Extensibilidad de grano fino

La solución del sistema SPIN es proporcionar una infraestructura para ejecutar el código del usuario dentro del núcleo (extensiones denominadas *spindles*). Se restringe estos *spindles* a que estén programados en un lenguaje especial seguro, un subconjunto del Modula-3. La utilización de este lenguaje seguro, la existencia en el sistema de un compilador verificado del mismo y un metalenguaje específico para el control en tiempo de enlace permite asegurar que los *spindles* no comprometan la seguridad del núcleo.

3.2 Mecanismos de Computación en Spin

Spin no define un modelo de hilos para las aplicaciones sino que define la estructura de soporte de una implantación de un modelo de hilos. Esta estructura se define mediante un conjunto de **eventos** que son gestionados por planificadores o paquetes de hilos.

Estos paquetes multiplexarán los recursos de procesamiento de bajo nivel entre contextos que compiten denominado **strands** (similar a un hilo dado que refleja un contexto del procesador). Cada paquete de hilos define la implantación de la interfaz del strand para sus propios hilos.

Así, en Spin una aplicación puede proporcionar su propio paquete de hilos que se ejecuta en modo kernel. Gracias a esto, Spin permite una gestión de hilos extensible.

3.3 Crítica

SPIN no es un sistema diseñado con el objetivo de la orientación a objetos, por tanto no da soporte directo a ningún elemento relacionado con la OO, sus abstracciones son convencionales. Existe una separación clara entre el espacio del usuario y el del sistema, aunque la característica principal de SPIN es permitir la inclusión de código de usuario en el núcleo del sistema para extenderlo.

3.3.1 Falta de uniformidad para la extensión

El problema de estas extensiones es que la manera de hacerlas es específica, no uniforme con la forma normal de utilización del sistema por el usuario. Las extensiones se programan con un lenguaje especial y se utiliza un metalenguaje también especial para enlazarlas con el núcleo. Esto es totalmente diferente de cómo se programan las aplicaciones normales de usuario.

3.4 Características interesantes

Como uno de los sistemas pioneros que permite extender dinámicamente el sistema, ésta es su característica más importante, junto con la necesidad de controlar esta extensión.

3.4.1 Extensibilidad dinámica por código de usuario

Es importante la posibilidad de que el usuario pueda extender dinámicamente la funcionalidad del sistema programando el mismo las extensiones. Esto da mucha flexibilidad al sistema. Para mantener la uniformidad, en el sistema integral esta extensión serán objetos del usuario, y se deberá realizar de manera uniforme y no diferente a la manera de programar normal en el sistema.

Cabe destacar la posibilidad de implementar entornos de computación apropiados para cada aplicación descargando código que defina la interfaz de los contextos y las reglas que rigen la relación entre los strands (contextos de ejecución a nivel usuario) y los recursos de procesamiento.

3.4.2 Seguridad en la extensibilidad

La extensibilidad es un mecanismo muy potente, pero que puede poner en riesgo el buen funcionamiento del sistema. Es importante controlar que el código de usuario no pueda salir de su ámbito y acceder o modificar servicios que no tenga por qué utilizar. En el caso del sistema integral se deberá controlar a los objetos que extiendan el sistema, siempre de manera uniforme con el resto del sistema y no mediante medios específicos.

3.4.3 Eficiencia

Una vez instalada la extensión, esta se ejecuta como parte del sistema operativo lo que hace que el coste de una llamada a una extensión proporcionada por el usuario se limite a una llamada a procedimiento.

4 Aegis: Aproximación a un Sistema Operativo Extensible

El sistema operativo Aegis[EKT95] pertenece a la categoría de los **Exokernels**. Las arquitecturas exokernel para los sistemas operativos proponen minimizar el código del núcleo del sistema operativo de forma que proporcione la mínima funcionalidad posible y proporcionar todos los mecanismos de gestión de los recursos físicos a nivel de aplicación.

4.1 Arquitectura del sistema

La arquitectura exokernel consiste en un núcleo mínimo que realiza únicamente tareas de muy bajo nivel.

- Se encarga de multiplexar y exportar los recursos físicos de forma segura valiéndose de un conjunto de primitivas de bajo nivel.
- Propaga las excepciones.
- Implementa la *upcalls* para ejecutar código a nivel usuario.

Todo esto proporciona a las aplicaciones la posibilidad de trabajar cerca del hardware gestionándolo a nivel de usuario con todas las posibilidades de gestión eficiente y flexible de los recursos.

El sistema operativo se construye en gran medida sobre el exokernel mediante librerías de sistemas operativos que usan la interfaz proporcionada por él e implementan abstracciones de más alto nivel. Así, pueden definir implantaciones de propósito especial que se adecuan mejor a los objetivos de rendimiento y funcionalidad de las aplicaciones.

Esta estructura permite a las aplicaciones la extensión, especialización e incluso el reemplazamiento de las abstracciones existentes.

4.2 Abstracción para la Computación en Aegis

Aegis representa la CPU como un vector lineal donde cada elemento se corresponde con un intervalo de tiempo de reloj o *time-slice*.

Estos intervalos son los recursos que el sistema ofrece para la ejecución. Las aplicaciones pueden solicitar intervalos de tiempo de CPU, de forma similar a como solicitan una página de memoria física, incluso es posible solicitar un conjunto de intervalos concretos para mejorar la latencia o el *throughput*.

La librería del sistema operativo que trabaja sobre la interfaz del exokernel, implementará las abstracciones de alto nivel y definirá la implantación específica que se ajuste mejor a los requerimientos de rendimiento y funcionalidad de una aplicación concreta.

De esta forma, es posible redefinir a nivel de aplicación las abstracciones más fundamentales de un sistema operativo.

4.3 Planificación en Aegis

Primitiva yield

El mecanismo de planificación de Aegis se basa en el uso de una primitiva de cesión directa del procesador, **yield**, que permite a los hilos de la aplicación planificar a otros hilos. Las aplicaciones pueden basarse en esta primitiva para implementar su planificador específico.

La implantación a nivel de librería mantiene una lista de los procesos de los que es responsable junto con el tiempo que les toca recibir del *time-slice*. Cada vez que un

intervalo termina, el planificador recalcula a qué proceso le toca y le cede el procesador a él directamente.

Interrupciones de Tiempo

Las interrupciones de tiempo denotan el comienzo y el fin de los intervalos de tiempo y se lanzan de forma similar a las excepciones. En el exokernel se realizan las tareas mínimas como cargar el contador de excepción del programa y saltar a ejecutar el código de gestión de la interrupción especificado. Mientras que son los gestores de interrupción específicos de cada aplicación los que se encargan del resto de las tareas como son el cambio de contexto.

4.4 Crítica

4.4.1 Mecanismo Complejo de Bajo Nivel para implementar la planificación

La combinación de los mecanismos de ejecución de Aegis con el requisamiento presenta algunos problemas. Si un hilo va a ser requisado, es necesario llamar a una rutina a nivel usuario que salve los registros del hilo. Dado que está implementada por la aplicación, se instala un *deadline* para ella. El problema es que puede tardar más tiempo que el *deadline*.

Una propuesta que solventa el problema, a costa de complicar el modelo, es asignar un hilo que se haga cargo del procesador y lo done al hilo que va a ser activado.

4.4.2 Mecanismo de muy bajo nivel, demasiado complejo para la mayor parte de las aplicaciones

No cabe duda de que, la mayor parte de las aplicaciones no utilizarán los mecanismos que ofrece Aegis para personalizar su entorno, fundamentalmente, debido a la complejidad inherente de su uso.

4.5 Cuestiones Interesantes

4.5.1 Flexibilidad

Pretende obtener flexibilidad y rendimiento exponiendo de forma segura las primitivas hardware de bajo nivel.

Las librerías de sistemas operativos que usan la interfaz proporcionada por el exokernel, implementan abstracciones de más alto nivel y pueden definir implantaciones de propósito especial que se adecuan mejor a los objetivos de rendimiento y funcionalidad de las aplicaciones.

Esta estructura permite a las aplicaciones la extensión, especialización e incluso el reemplazamiento de las abstracciones existentes.

4.5.2 Eficiencia

Al eliminar completamente las abstracciones de los sistemas operativos tradicionales inadecuadas para algunas aplicaciones y permitir que sean éstas las que construyan las que mejor se adapten a sus necesidades, se mejora notablemente la eficiencia.

Adicionalmente, es posible que coexistan múltiples implantaciones de una misma abstracción cada una de las cuales será más adecuada que las demás para proporcionar determinada funcionalidad o rendimiento.

5 Apertos

Apertos[Yok92], una evolución de su predecesor Muse [YTY+89] suele considerarse como el pionero en la aplicación de la reflectividad en los sistemas operativos orientados a objetos.

Es un sistema operativo para dar soporte a objetos y estructurado uniformemente en términos de objetos. La motivación inicial de aplicación es para un entorno de computación móvil, compuestos por ordenadores que pueden estar conectados a la red y también desconectarse para trabajar independientemente o trabajar remotamente mediante enlaces sin hilos, etc. La gran novedad de este sistema consiste en utilizar una separación de los objetos en dos niveles: meta-objetos y objetos base.

Estructuración mediante objetos y meta-objetos

Los **meta-objetos** proporcionan el entorno de ejecución y dan soporte (definen) a los **objetos base**. Cada objeto base está soportado por un **meta-espacio**, que está formado por un grupo de meta-objetos. Cada meta-objeto proporciona una determinada funcionalidad a los objetos del meta-espacio en que se encuentra. Por ejemplo, un determinado objeto puede estar en un meta-espacio con meta-objetos que le proporcionen una determinada política de planificación, un mecanismo de sincronización determinado, el uso de un protocolo de comunicación, etc.

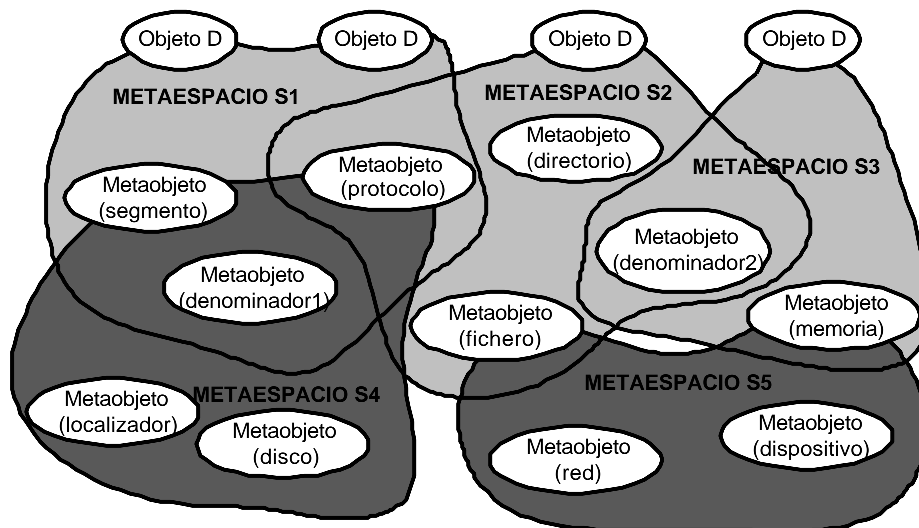


Figura 4.5 Separación entre los espacios de objetos y de meta-objetos en el sistema Apertos.

Flexibilidad

La flexibilidad del sistema se consigue mediante la posibilidad de que un objeto cambie de meta-espacio, o la adición de nuevos meta-objetos que proporcionen funcionalidad adicional. Por ejemplo, un meta-objeto podría migrar de un meta-espacio dado a otro meta-espacio que use un protocolo de comunicaciones para una red sin hilos cuando el ordenador en el que se encuentre se desconecte físicamente de la red.

Reflectividad

La reflectividad se produce al existir una interfaz bidireccional entre los objetos base y los meta-objetos que los soportan. Los objetos pueden dialogar con sus meta-objetos, usando un punto de entrada al meta-espacio denominado **reflector**. A su vez, los meta-objetos influyen en el funcionamiento de los objetos base. Por otro lado ambos, objetos y meta-objetos, comparten el mismo marco conceptual al estar descritos en los mismos términos de objetos. Los meta-objetos también pueden considerarse como objetos, por lo que tendrían su propio meta-meta-espacio y así sucesivamente. Esta regresión termina en un meta-objeto primitivo que no tiene meta-meta-espacio (se describe a sí mismo).

Los objetos y meta-objetos se organizan en una jerarquía de clases, al igual que los reflectores. Apertos se estructura usando una serie de jerarquías predefinidas de reflectores y meta-objetos.

Jerarquía de reflectores

La jerarquía de reflectores define la estructura común de programación de meta-objetos del sistema. Un reflector, como punto de entrada a un meta-espacio, ofrece a los objetos base una serie de operaciones que pueden usar de sus meta-objetos. La jerarquía base define las operaciones comunes del sistema. El reflector mCommon contiene una operación común a todos los reflectores, canSpeak que permite comprobar la compatibilidad entre meta-espacios a la hora de la migración de los objetos. Otros reflectores son mRealtime, que ofrece a los objetos una planificación en tiempo real, mDriveObject, que proporciona meta-operaciones para programación de controladores de dispositivos, etc.

MetaCore es un meta-objeto terminal en el que finaliza la regresión de meta-meta-objetos. Puede considerarse como el equivalente de un micronúcleo de otros sistemas. Proporciona a todos los demás objetos del sistema con los elementos básicos del sistema, es decir, las primitivas comunes de separación entre objetos y meta-objetos y de migración de objetos. Proporciona el concepto de contexto de ejecución para virtualizar la CPU y las primitivas básicas de la computación reflectiva: M hace una petición de meta-computación (llamada al metaespacio). R reanuda la ejecución de un objeto (retorno de la llamada al metaespacio).

5.1 Crítica

Este sistema sí que utiliza como única abstracción los objetos y da soporte directo a los mismos. Existen, sin embargo, algunos aspectos que no están totalmente de acuerdo con los objetivos del sistema integral. Por ejemplo no hay previsión directa para la portabilidad.

5.1.1 Complejidad de estructura

La separación de la estructura del sistema en múltiples meta-espacios recursivos, aunque da mucha flexibilidad al sistema, también complica la comprensión del mismo de manera sencilla por el usuario.

5.1.2 Falta de uniformidad por la separación espacio/meta-espacio de objetos

La separación completa de ambos espacios introduce una cierta falta de uniformidad en el sistema. La jerarquía de meta-objetos está totalmente separada de la de los objetos de usuario. La programación de meta-objetos se realiza por tanto de una manera diferente a la de los objetos normales.

5.1.3 No existe mecanismo de seguridad uniforme en el sistema

Muchos elementos del sistema no se definen, y se deja que se implementen mediante meta-objetos. Sin embargo, en casos como la seguridad, es conveniente para la uniformidad y la fácil comprensión del sistema que exista un mecanismo básico y uniforme de protección, que forme parte integrante del núcleo del sistema.

5.2 Características interesantes

La característica principal es la utilización de la reflectividad como elemento básico del sistema operativo.

5.2.1 Reflectividad para la flexibilidad

La adopción de la reflectividad en el sistema es muy importante para lograr la flexibilidad en el sistema de manera uniforme. Esta propiedad permite describir mediante objetos los propios elementos de la máquina. De esta manera se unifican dentro del paradigma de la OO los elementos del usuario y los elementos del sistema que los soportan. Esto permite la extensibilidad del sistema de una manera uniforme, al permitir que los objetos de usuario puedan acceder a los objetos del sistema usando el mismo paradigma de la OO.

6 Máquina Virtual Java

La **especificación de la máquina virtual Java**[Sun97a, Ven98], describe el comportamiento de una instancia de la JVM en términos de subsistemas, áreas de memoria, tipos de datos e instrucciones. Estos componentes describen la arquitectura interna para la JVM.

El propósito de todos estos componentes no es tanto dictar la arquitectura interna a seguir en una implantación, sino proporcionar un modo de definir, estrictamente, el comportamiento externo de las implantaciones.

La especificación define el comportamiento requerido por cualquier implantación de la JVM en términos de estos componentes abstractos y su interacción.

En la siguiente figura se muestra un diagrama de bloques de la JVM que incluye las áreas y subsistemas más significativos y que se describen a continuación.

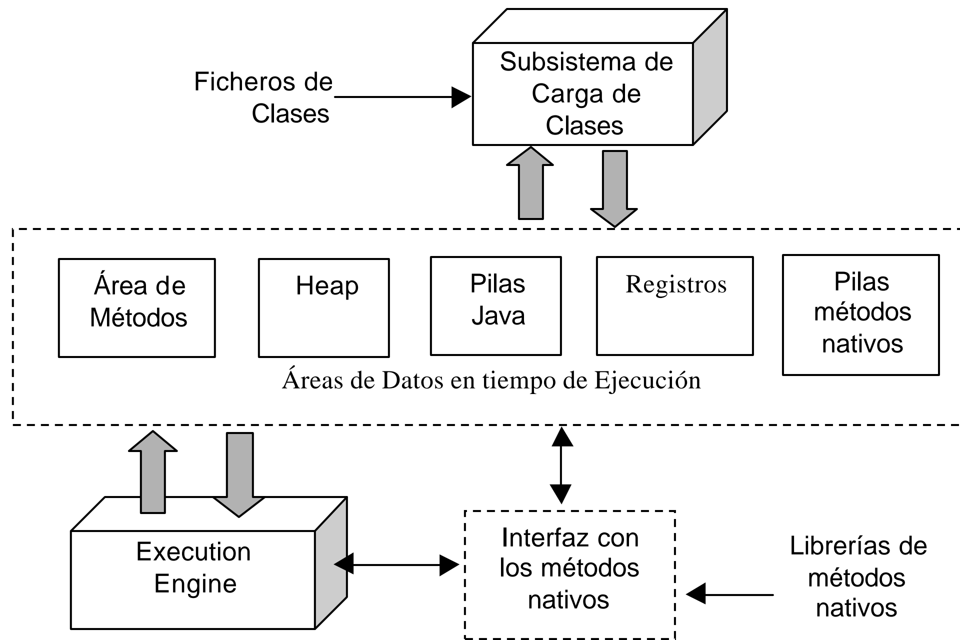


Figura 4.6 Arquitectura Interna de la Máquina Virtual Java.

6.1 Subsistema de Carga de Clases

Se trata de la parte de la implantación de la JVM que se encarga de llevar a cabo la carga de clases:

- **Carga** (*loading*). Localiza el fichero apropiado e importa los binarios – leer el fichero y pasarlo a la máquina virtual.
- **Enlace** (*linking*). Verifica la corrección de las clases importadas, asigna e inicializa memoria para las variables de clase y ayuda en la resolución de las referencias simbólicas
- **Inicialización**. Invoca el código Java que inicializa las variables de clase a sus valores iniciales apropiados.

6.2 Runtime Data Areas

Cuando la JVM ejecuta un programa, necesita memoria para almacenar no sólo *bytecodes*, sino también otra información extraída de los ficheros de clases cargados, objetos que el programa instancia, parámetros de los métodos, variables locales, etc.

La JVM organiza la memoria que necesita para ejecutar un programa en varias **áreas de datos en tiempo de ejecución**, del inglés *runtime data areas*.

6.2.1 Área de Métodos

Se trata de un área de memoria en la que se almacena información acerca de los tipos cargados. Cuando el cargador lee el fichero de clases, pasa la información a la

máquina que extrae información acerca del tipo y la almacena en el área de métodos, al igual que las variables de clase.

6.2.2 Heap

Siempre que se crea una instancia de una clase, o un array en una aplicación Java en ejecución, la memoria necesaria para el nuevo objeto se asigna del heap. Cada aplicación que se ejecuta corre sobre su propia y exclusiva máquina Java, por lo que no comparten el heap, sin embargo, los hilos dentro de una misma aplicación, sí comparten el heap.

6.2.3 Pila

Cuando se crea un nuevo hilo, la máquina Java crea para él una nueva pila Java que se encarga de almacenar la información del hilo en capas. Se habla de método actual, capa actual, clase actual y constant pool actual, para designar al método actualmente ejecutado por el hilo, la capa para el método actual, la clase actual y el pool de constantes de dicha clase.

Cuando un hilo invoca un método Java, la máquina virtual crea y apila una capa en la pila, que pasa a ser la capa actual. Esta capa se utiliza para almacenar parámetros, variables locales, etc.

Un método termina normal o abruptamente. En cualquier caso, la máquina Java desapila y descarta la capa de la pila correspondiente a la ejecución de ese método.

6.2.4 Registros

Cada hilo en ejecución en un programa tiene su propio registro contador de programa, creado cuando se arranca el hilo.

A medida que el hilo ejecuta un método, el PC contiene la dirección de la instrucción actualmente ejecutada.

6.3 Execution Engine en Java

El corazón de una implantación cualquiera de la JVM es el **entorno de ejecución**, del término inglés *execution engine*.

La especificación abstracta que define su comportamiento en términos de un conjunto de instrucciones. Para cada instrucción, la especificación describe en detalle qué debe hacer la implantación cuando encuentra esa instrucción.

Una instancia en tiempo de ejecución de un entorno de ejecución es un **thread**. Cada hilo de una aplicación Java es una instancia distinta e individual del entorno de ejecución de una máquina virtual. Desde el principio de su vida, el hilo ejecuta bytecodes o métodos nativos.

La máquina Java es una **máquina multihilos**, donde los hilos son elementos de la implantación de la máquina, internos. La especificación de la máquina virtual Java define un modelo de hilos cuyo propósito es facilitar la implantación en una amplia variedad de arquitecturas. Uno de sus objetivos es permitir a los diseñadores de implantaciones usar hilos nativos donde sea posible.

6.3.1 Planificación

El modelo de hilos de Java especifica que un hilo puede ejecutarse con una prioridad entre 10 siendo 1 la menor y 10 la mayor. La especificación de la máquina determina el comportamiento de los hilos en diferentes prioridades únicamente diciendo que todos los hilos de mayor prioridad conseguirán la CPU, mientras que los de prioridad más baja sólo la conseguirán cuando los más prioritarios se bloqueen.

La especificación no asume tiempo-compartido entre los hilos de diferentes prioridades.

6.3.2 Sincronización

La especificación de la JVM sí indica que cualquier implantación debe soportar, en la implantación de los hilos, dos aspectos de la sincronización: bloqueo de objetos y wait y notify de los hilos.

Bloqueo de objetos

Ayuda a evitar que los hilos interfieran unos con otros en el acceso de un objeto compartido.

Wait y Notify

Ayuda a los hilos a cooperar unos con otros en la consecución de un objetivo común.

6.4 Características Interesantes

Son muchas las características interesantes que se derivan de la especificación de la Máquina Virtual de Java.

6.4.1 Independencia de la plataforma

Una de las razones más importantes para el éxito de Java es la posibilidad que ofrece de poder crear binarios ejecutables en distintas plataformas. Si se tiene en cuenta el entorno actual, formado por una maraña de máquinas heterogéneas, se comprenderá que esta sea una muy buena razón para optar por un diseño que proporcione esta característica.

6.4.2 Soporte Básico a los objetos

La máquina virtual Java define un modelo de objetos que será heredado por los lenguajes, entornos de programación, Bases de Datos, etc. que se ejecuten sobre ella.

Además de esto, proporciona el soporte básico para la identificación, almacenamiento y ejecución de los objetos.

Estas características facilitan la creación de un marco común de definición y ejecución de objetos que conlleva la eliminación de los problemas de interoperabilidad señalados en el Capítulo I.

6.4.3 Soporte básico a la Computación

La JVM especifica que cualquier implantación de la máquina debe ofrecer soporte a la ejecución de los métodos de los objetos de forma interna. Esto evita el que tengan que definirse abstracciones diferentes al objeto, que deban ser manejadas a alto nivel[1].

6.4.4 Máquina Multihilo

La especificación de la JVM determina que esta es una máquina multihilo, lo que garantiza un buen rendimiento, sobre todo en entornos multiprocesadores.

6.5 Crítica

6.5.1 Falta de Definición de una Relación clara y estable entre los objetos y el entorno de ejecución de la máquina

La carencia más reseñable de la máquina virtual de Java es la ausencia de una relación clara entre los objetos y los hilos que ejecutan sus métodos. Esto es un hándicap importante a la hora de implantar un mecanismo sencillo que permita migrar los objetos juntamente con los hilos que están ejecutando sus métodos, a través de una red.

➔ **Trabajos como ¿?? señalan tal dificultad. Fer tenía algún artículo de esto.**

[1] Esto no quiere decir que los lenguajes no puedan definir esas otras abstracciones si lo desean. Únicamente, deberán construirlas sobre la abstracción de objeto ofrecida por la JVM.

Capítulo V

Un Sistema Integral Orientado a Objetos

En los capítulos anteriores se ha definido un problema, la construcción de un entorno OO que elimine los problemas de desadaptación entre capas, de muy difícil solución. Algunos de los sistemas OO revisados en el capítulo IV dan una solución parcial a estos problemas, aunque ninguno de ellos da una solución integral al problema.

Una de las mayores dificultados en el diseño y construcción del entorno OO es la integración de la computación en el modelo de objetos de forma homogénea, sin necesidad de añadir abstracciones diferentes al objeto que compliquen el modelo.

En lo que sigue se abordarán todos estos problemas dando soluciones a cada uno de ellos. En este capítulo se identificará la construcción de un Sistema Integral Orientado a Objeto (SIOO), formado por una Máquina Abstracta Orientada a Objeto (MAOO) y un Sistema Operativo Orientado a Objeto (SOOO) como el marco más adecuado para el soporte a objetos.

En los capítulos posteriores se solucionarán los problemas de cómo combinar ambas piezas (Capítulos XII y XIV) y la definición del modelo de objetos para la computación (Capítulos VII, VIII, IX y X).

1 Sistema Integral Orientado a Objetos como Solución Global

Los problemas de integración identificados en el capítulo I se derivan fundamentalmente de la falta de una abstracción uniforme en todas las capas del sistema con los cambios de paradigma y la merma en la productividad que eso supone.

Una manera de resolver este problema es construir un **sistema homogéneo** que utilice en todos sus elementos el paradigma de la orientación a objetos y dé soporte nativo a los mismos, es decir, se trata de diseñar y construir un **Sistema Integral Orientado a Objetos** (SIOO).

A continuación se describe un SIOO que da solución al problema de integración y cuya primera versión recibió el nombre de Oviedo3[CIA+97], proyecto de investigación que pretende construir un sistema experimental basado en este principio.

Una descripción general del SIOO, su arquitectura y el modelo de objetos puede encontrarse en [ATA+98].

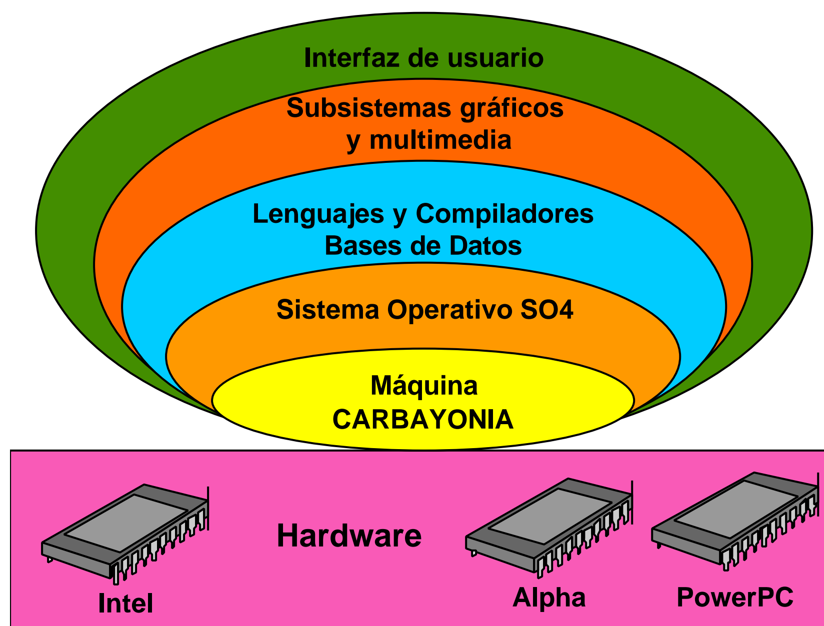


Figura 5.1 Esquema general del sistema integral orientado a objetos Oviedo3

1.1 Entorno Integral Orientado a Objetos

Un entorno integral Orientado a objetos consiste en aplicar o adoptar el paradigma de orientación a objetos en todas las fases de análisis, diseño e implantación así como en todas las capas que componen un sistema de computación: lenguajes, bases de datos, etc.

El sistema conoce y comprende el concepto de objeto y ofrece medios para gestionarlo.

Elimina la Desadaptación

Al usar el mismo paradigma en todas las capas que componen el sistema, **no existe desadaptación**. Desde una aplicación OO, tanto el acceso a los servicios del sistema como la interacción con otros objetos se realizan utilizando los mismos términos OO.

Soluciona el problema de Interoperabilidad

Al dar soporte directo a objetos, es decir, al gestionar directamente tanto la representación de los objetos como su utilización se **soluciona el problema de la interoperabilidad**. Los objetos no son conceptos que sólo existen dentro de los procesos del SO y que sólo son conocidos por el compilador que creó el proceso. Al ser gestionados por el propio sistema pueden "verse" unos a otros, independientemente del lenguaje y del compilador que se usó para crearlos.

Reduce la Complejidad

No son necesarias capas de adaptación con lo que **se reduce la complejidad conceptual y técnica del sistema**. Los usuarios/programadores se ven liberados de preocuparse por detalles técnicos y contraproductivos cambios de paradigma y pueden

concentrarse en el aspecto más importante que es la resolución de los problemas usando la OO.

La construcción de un Sistema Integral Orientado a Objetos en el que se aplique únicamente el paradigma de orientación a objetos en todas las capas del sistema se manifiesta pues como un mecanismo adecuado para solucionar los problemas de integración e interoperabilidad entre las distintas capas de un sistema. Se pueden citar varios antecedentes de esta idea, como [Sch96].

1.2 Soporte al Objeto en el Sistema Integral

Para constituirse en solución a los problemas de la adopción del paradigma de OO, un Sistema Integral debe ofrecer **soporte nativo a los objetos** en la capa más básica, de forma que todas las capas superiores conozcan y adopten el objeto como abstracción fundamental y el paradigma de OO como "medio de organización".

La integración del concepto de objeto, aún en las capas más bajas del sistema, proporciona una serie de ventajas que se transforman en una integración uniforme y eficiente del paradigma de orientación a objetos en todas las capas del sistema.

Hace posible la existencia de objetos como entidad

Al proporcionar los mecanismo de representación y gestión de objetos en la capa más básica del sistema, el sistema completo, desde los lenguajes de programación a los sistemas operativos, incluyendo la máquina subyacente, consistiría en un conjunto de objetos identificables y que se comunican entre sí para la realización de tareas[Sch96].

Facilita la interacción de cualesquiera Objetos

El hecho de que se de soporte a objetos los objetos "aparecen en sociedad", son conscientes de su existencia y de la de los otros objetos y pueden interaccionar unos con otros, solucionando los problemas de interoperabilidad presentados en el capítulo I.

Hasta ahora sólo existían dentro de los procesos tradicionales de los sistemas operativos y sólo eran conocidos por el compilador que creó el proceso.

Modelo de Objetos Común

Existe un modelo de objetos común, el ofrecido por la capa más básica del sistema y heredado por las capas superiores. Todos los elementos del sistema conocen y entienden este modelo.

1.3 Sistema = Mundo de Objetos

Un sistema integral orientado a objetos ofrece al usuario un entorno de computación que crea un mundo de objetos: *un único espacio de objetos, virtualmente infinito, en el que un conjunto de objetos homogéneos coopera intercambiando mensajes independientemente de su localización, y donde los objetos residen indefinidamente hasta que ya no son necesitados.*

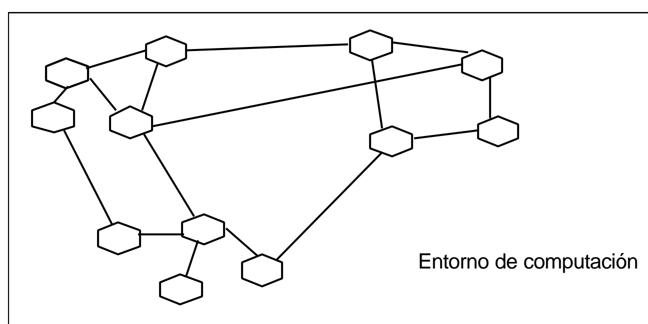


Figura 5.2 Entorno de computación compuesto por un conjunto de objetos homogéneos.

De esta forma, se ofrece al usuario un entorno de computación integral OO en el que se aprovechan las ventajas del paradigma OO como la reusabilidad[ATA+98].

Al utilizar un paradigma común se logra que todas las capas del sistema "hablen el mismo idioma" o lo que es lo mismo, manejen los mismos conceptos, evitándose así el costoso paso de la traducción.

Esta solución presenta múltiples ventajas derivadas de la adopción del paradigma de OO especialmente en áreas como la flexibilidad, extensibilidad y reusabilidad que se verán en capítulos posteriores dedicados a los componentes del SIOO.

2 Requisitos de diseño de un Sistema Integral Orientado a Objetos

Los principales aspectos de diseño[ATA+98] que se tendrán en cuenta para el diseño de un entorno como el mencionado son la necesaria uniformidad del modelo de objetos y la homogeneidad, manteniendo la sencillez en el diseño.

2.1 Modelo de objetos Uniforme.

En un sistema como el deseado, la única entidad existente en el sistema será el objeto, independientemente de su granularidad. Es decir, se ofrecerá una única abstracción, y no existirán diferencias entre objetos del lenguaje o del sistema[YMF+91].

El **modelo de objetos es intencionadamente estándar**, con las características más comunes de los lenguajes de programación más extendidos: encapsulación, herencia y polimorfismo[Boo94].

Guiados por el objetivo de conseguir un modelo de objetos lo más "estandarizado" posible, se escogen las características más utilizadas en los lenguajes de programación populares y en general más aceptadas entre la comunidad OO, junto con las características adicionales descritas anteriormente.

Así, las características fundamentales de este modelo único de objetos son las que a continuación se detallan[Alv97].

- **Abstracción y encapsulamiento**
  - Identidad única de objetos
  - Clases
- **Jerarquía**
  - Herencia múltiple (es-un)
  - Agregación (es-parte-de)
- Relaciones generales de asociación (asociado-con)
- **Modularidad**
- **Tipos**
- Comprobación de tipos, incluso en tiempo de ejecución
- Polimorfismo
- **Excepciones**
- **Concurrencia**
- **Persistencia**
- **Distribución**
- **Seguridad**

2.1.1 Abstracción y Encapsulamiento: Clases

Estas propiedades suelen describirse como un conjunto. La **abstracción** se define en [Boo91, Boo94] como sigue:

"Una abstracción denota las características coincidentes de un objeto[1] *que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador".*

Es decir, un objeto presenta un comportamiento esencial expuesto a través de una interfaz o conjunto de métodos.

Esta propiedad se suele combinar con la de **encapsulamiento**, que establece el principio de ocultación de información.

"El encapsulamiento es el proceso de almacenar en un mismo compartimento los elementos de una abstracción que constituyen su estructura y su comportamiento, sirve para separar la interfaz contractual de una abstracción de su implementación".

Es decir, la implantación de los métodos y estado interno de un objeto debe estar totalmente encapsulada en el mismo y separada de su interfaz y la única manera de utilizar un objeto es mediante los métodos de su interfaz.

También puede considerarse que el objeto encapsula en general toda su semántica, incluyendo las propiedades implícitas que puedan existir en el modelo de objetos, como la concurrencia, que se describen más adelante.

---

[1] Se utilizarán los nombres objeto e instancia indistintamente

Clases

Como unidad de abstracción y representación del encapsulamiento, la mayoría de los sistemas OO utilizan el concepto de **clase**, que describe las características que tendrán todos los objetos de la misma: la representación del estado interno y su interfaz.

Identidad única de objetos

En un SIOO homogéneo en el que todos los objetos están al mismo nivel, tiene que existir un mecanismo que permita asignar un identificador a cada objeto. Este identificador debe ser único en todo el SIOO, para que el sistema distinga un objeto de otros.

Este identificador único, al permitirse la distribución de objetos, debe ser transparente a la localización, debe ser global e identificar de manera unívoca cada objeto dentro del sistema hasta que el objeto desaparezca.

Este identificador único será la única manera de acceder al objeto dentro del sistema. Como se verá, la utilización de un **identificador único** para los objetos facilita la realización de propiedades, fundamentalmente, la persistencia y la distribución, de manera transparente.

2.1.2 Jerarquía. La relación "es-un" (herencia). La relación "es-parte-de" (agregación)

La **jerarquía** es "*una clasificación u ordenación de abstracciones*". Permite comprender mejor un problema agrupando de manera jerárquica las abstracciones (objetos) en función de sus propiedades o elementos comunes, descomponiendo en niveles inferiores las propiedades diferentes o componentes más elementales.

Herencia. La relación "es-un"

Es una relación que se establece entre las diferentes clases de un sistema. Esta relación indica que una clase (**subclase**[2]) comparte la estructura de comportamiento (las propiedades) definidas en otra clase (**superclase**[3]). La clase "hereda[4]" las propiedades de la superclase. Se puede formar de esta manera una jerarquía de clases[5]. Un objeto de una clase también "es-un" objeto de la superclase (por ejemplo, un coche es un vehículo). En el caso en que una clase pueda heredar de más de una superclase se habla de **"herencia múltiple"** por contraposición al caso de **"herencia simple"**.

Agregación. La relación todo/parte ("es-parte-de")

Las jerarquías "es-parte-de" describen relaciones de agregación entre objetos para formar otros objetos de un nivel superior. Permiten definir un objeto en términos de agregación de sus partes, de objetos componentes más elementales (ALU, registros, ..., son parte de un procesador).

---

[2] O clase hija.

[3] O clase padre.

[4] O "deriva de".

[5] Las clases situadas por encima de una clase en la línea de herencia son los ancestros, antepasados o ascendientes. Las que derivan de ella son los descendientes.

Relaciones generales de asociación

En un sistema, además de las relaciones "es-un" y "es-parte-de", existen más relaciones entre objetos. Es conveniente conocer qué objetos están relacionados con otros. Esta información puede ser útil cuando se tienen en cuenta aspectos de distribución y persistencia de los objetos. Por ejemplo, al mover un objeto a otro ordenador, es interesante mover también los objetos que están relacionados, puesto que es muy probable que se intercambien muchos mensajes entre sí.

La relación "es-parte-de", que permite conocer los objetos integrantes de otro objeto y moverlos a la vez a almacenamiento secundario o a otra máquina, se manifiesta insuficiente ya que no todas las relaciones pueden modelarse como "es-parte-de".

Si no existe otro tipo de relación en el modelo, el sistema no tendría conocimiento de esta relación y no podría explotarlo al igual que lo puede hacer con la relación "es-parte-de".

Como compromiso, se añade un tercer tipo de relación al modelo de objetos: la **relación de asociación** "asociado-con", que represente en general la existencia de otras relaciones diferentes a las de herencia o agregación entre objetos.

2.1.3 Modularidad

La **modularidad**[6] permite fragmentar un problema complejo en una serie de conjuntos de objetos o módulos, que interactúan con otros módulos. En cierta manera, representa un nivel superior al nivel de encapsulamiento, aplicado a un conjunto de objetos.

2.1.4 Tipos y polimorfismo

"Los tipos son la puesta en vigor de la clase de los objetos, de manera que los objetos de tipos distintos no pueden intercambiarse, o, como mucho, pueden intercambiarse sólo de formas muy restringidas". En realidad un **tipo** denota simplemente una estructura común de comportamiento de un grupo de objetos. Normalmente se identifica el concepto de tipo con el de clase[7], haciendo que la clase sea la única manera de definir el comportamiento común de los objetos. De esta manera la jerarquía de tipos se funde con la jerarquía de clases. Así un objeto de un subtipo (subclase) determinado puede utilizarse en cualquier lugar en el que se espera un objeto de sus tipos ancestros (clases ancestras).

Enlace estático y dinámico. Polimorfismo

El enlace hace referencia al momento en el que un nombre o referencia se asocia con su tipo.

En el **enlace estático** esto se realiza en tiempo de compilación. De esta manera la invocación de una operación sobre un objeto siempre activa la misma implementación de la operación.

---

[6] Un excelente tratamiento sobre la modularidad y la orientación a objetos en general se encuentra en [Mey88, Mey97].

[7] En este trabajo se asociará siempre el concepto de tipo con el de clase, salvo indicación explícita de lo contrario.

En el **enlace dinámico**, el tipo del objeto se determina en tiempo de ejecución y el sistema determinará qué implantación de la operación se invocará, en función de cuál sea el objeto utilizado.

Esto normalmente se denomina **polimorfismo** puesto que podemos aplicar el mismo nombre de operación a un objeto y en función de cual sea este objeto concreto el sistema elegirá la implantación adecuada de la operación.

La combinación de polimorfismo y herencia establece un compromiso entre la comprobación de tipos en tiempo de ejecución y la flexibilidad del enlace dinámico. De esta manera se puede usar el polimorfismo en un árbol de herencia compartiendo el mismo nombre de operación para las superclase y sus subclases, pero restringiendo la utilización de las operaciones a las que se pueden aplicar sobre el objeto (las de su árbol de herencia), siempre que pertenezca a una de las clases de la jerarquía.

2.1.5 Excepciones

Las excepciones son un concepto orientado fundamentalmente a la realización de programas robustos mediante el manejo estructurado de los errores. Básicamente, una **excepción** es un evento que se produce cuando se cumplen ciertas condiciones (normalmente condiciones de error). En estos casos se "lanza una excepción". La excepción será "atrapada" por un manejador de excepciones que realizará el tratamiento adecuado a la misma.

Las excepciones son un concepto importante con soporte en lenguajes populares como C++ y Eiffel. Parece adecuado, pues, introducirlo en el modelo de objetos del sistema para promover el desarrollo de programas robustos. También permite la integración fluida en el modelo de objetos de ciertas tareas fuera del ámbito de la programación típica, pero que debe realizar un sistema integral, como la gestión de interrupciones de dispositivos, etc. que pueden asimilarse a excepciones.

2.1.6 Concurrencia

"La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo".

La mayoría de los lenguajes OO están pensados para sistemas tradicionales con conceptos como fichero y proceso. En estos casos, los objetos se comportan como entidades pasivas manipuladas según marque el flujo de ejecución secuencial del programa.

Sin embargo, entre las propiedades de un objeto está la de la **concurrencia**, es decir, la posibilidad de tener actividad independientemente del resto de los objetos. Esto permite describir sistemas de una manera más cercana a la realidad, en la que los objetos en muchos casos tienen funcionamiento independiente.

Este será un punto fundamental en esta tesis que tendrá un profundo tratamiento en los Capítulos VII, VIII, IX y X, donde se desarrollaran las líneas de diseño del entorno de computación concurrente del SIOO. Posteriormente, en los Capítulos X y XI se mostrará el diseño del entorno propuesto en esta tesis.

2.1.7 Persistencia

"Es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado)".

Para que los sistemas OO sean más cercanos a la realidad, es necesaria la **persistencia**, es decir, un objeto no tiene por qué dejar de existir por el simple hecho de la finalización del programa que lo creó. El objeto representa una abstracción que debe permanecer como tal hasta que ya no sea necesaria su existencia dentro del sistema. Hay que hacer notar que no sólo los datos de un objeto deben persistir, la clase a la que pertenece también debe hacerlo para permitir la utilización del objeto con toda su semántica.

Un extenso trabajo acerca de la introducción de la persistencia en un SIOO puede encontrarse en [Alv97, Ort97].

2.1.8 Distribución

Aunque no denominada como tal, la propiedad de la **distribución** es referida con el término de **persistencia en el espacio**. Un objeto puede moverse (en el espacio) por los distintos ordenadores que conformen el sistema de computación, manteniendo todas sus propiedades como en el caso de la persistencia. Es decir, los objetos son distribuidos.

En [ATD+98a, ATD+98b] se puede obtener más información acerca de la introducción de la propiedad de distribución en un SIOO.

Como se ve, ciertos objetivos del sistema integral orientado a objetos, como la concurrencia, la persistencia y la distribución, ya se encuentran reconocidos como parte fundamental del modelo de objetos. Sin embargo, la mayoría de los lenguajes y sistemas no suelen contar estas propiedades dentro de sus modelos de objetos.

2.1.9 Seguridad

El mecanismo de seguridad, fundamental en un SIOO, permite la protección de los recursos del sistema, en este caso de los objetos.

Más concretamente se trata de discernir de alguna forma los derechos de utilización de los métodos de un objeto por parte de otros objetos, posibilitando así que sólo ciertos objetos puedan invocar métodos de otros objetos y en el caso de cada objeto qué métodos concretos puede invocar.

2.2 Homogeneidad

Es un punto central en el diseño del sistema. Todos los objetos deben tener las mismas propiedades y el mismo tratamiento. No debe haber objetos especiales e incluso los objetos del sistema no son diferentes a cualquier otro objeto definido por el usuario.

Y, dado que los objetos, de cualquier granularidad, son la única abstracción existente en el sistema, el **modo de trabajo es exclusivamente OO**. Esto supone que las operaciones permitidas serán aquellas que se puedan realizar sobre objetos,

incluyendo en este apartado a las clases. Así, los objetos pueden crear clases que hereden de otras, crear objetos a partir de una clase y enviar mensajes a otros objetos.

2.3 Simplicidad

El diseño del SIOO debe mantenerse lo más fiel posible a las líneas anteriores, buscando la máxima sencillez.

3 Arquitectura del Sistema Integral

La **arquitectura propuesta**[ATA+98] para la construcción de un sistema integral orientado a objetos que cumpla los requisitos descritos en el apartado anterior, se basa en estructurar el sistema en torno a una **máquina abstracta orientada a objetos (MAOO)** y un **sistema operativo orientado a objetos(SOOO)**.

En concreto, el SIOO está compuesto, en su núcleo más básico por una MAOO, que actuaría como micronúcleo del sistema integral, proporcionando soporte para un modelo de objetos único en el sistema.

Sobre esta máquina abstracta se construye un SOOO que extiende y complementa el comportamiento de la máquina abstracta gracias a una arquitectura reflectiva, lo que proporciona una gran flexibilidad al entorno resultante. Este Sistema Operativo se construye como un conjunto de objetos, no distinguibles de otros objetos del sistema, que podrían haber sido definidos por el usuario, y que ofrecen la funcionalidad del sistema operativo, extendiendo la máquina abstracta en campos como la distribución, la concurrencia, la persistencia y la seguridad, de forma transparente. Juntos proporcionarán el espacio de objetos del sistema integral.

En capítulos posteriores se proporciona una descripción detallada de ambos. Concretamente, en el capítulo XV se describe la arquitectura y comportamiento de la máquina y en el Anexo A se describe el ensamblador de la misma. Por su parte, en el Capítulo XVI se describe el Sistema Operativo. En el capítulo XVII, se describe un prototipo de ambos.

Capítulo VI

El Sistema Operativo Orientado a Objeto: El Cerebro del Sistema Integral

A la hora de construir un nuevo sistema operativo que tiene como requisito global salvar el salto semántico existente actualmente entre los Sistemas Operativos tradicionales y las capas software OO que se ejecutan por encima, es innegable que la arquitectura interna más prometedora es la integración del paradigma de OO en el diseño y construcción del Sistema Operativo, tal y como ya se apuntaba en el capítulo II.

De hecho, la combinación de objetos y sistemas operativos es un área muy en boga actualmente, ya que la integración del paradigma de orientación a objetos en el campo de los sistemas operativos está todavía en una fase temprana de investigación e implantación de prototipos.

Sin embargo, y como ya se apuntó en los Capítulos II y III, esta integración no es un camino sencillo y muchas de las propuestas que actualmente se citan como SOOO no son más que la aplicación del paradigma de OO como herramienta para construir una plataforma software grande y compleja, con todas las ventajas derivadas de ello.

Pero no integran el concepto de objeto en la plataforma. Por decirlo de forma rápida, los objetos no existen en tiempo de ejecución, sólo en tiempo de diseño.

1 El Paradigma de OO y los Sistemas Operativos. ¿Falacia o Realidad?

Un **sistema operativo orientado a objetos** es mucho más que un Sistema Operativo implementado usando técnicas de orientación a objetos.

Se han planteado distintas teorías para la integración de objetos en los sistemas operativos que se diferencian en el grado y la forma de integración [Kra93, GKS94, Rus91b, Cah96b] y de las que ya se ofreció una muestra en el Capítulo II.

Áreas de Diseño

En un primer nivel de abstracción, existen fundamentalmente dos áreas en las que se centra el diseño de los sistemas operativos orientados a objetos.

1. Determinar qué nivel de soporte dará el sistema operativo a los objetos, no sólo a los que forman parte de sí mismo, sino también a cualquier otro objeto que pudiese ser definido por el usuario o por alguna otra capa software[Kra93].

2. Determinar la conveniencia de diseñar el propio sistema operativo como un conjunto de objetos que interaccionan entre sí.

Un Mundo de Objetos: Soporte a la abstracción de objeto en tiempo de ejecución

El alto grado de abstracción que arrastra el concepto de objeto y que contagia a la infraestructura o plataforma que debe darle soporte, obliga a que el tratamiento deba realizarse con sumo cuidado.

Para que un Sistema Operativo se pueda definir como Orientado a Objeto, es necesario que soporte la abstracción de objeto como componente estructural básico, apta para ser utilizada como elemento en la construcción de aplicaciones de usuario, con una semántica bien definida, a la vez que ofrece todos aquellos mecanismos necesarios para dar vida a tales entidades, como pueden ser la identificación, ejecución de sus métodos o comunicación e interacción entre ellos.

Así pues, el sistema operativo debe proporcionar el concepto de objeto, no como elemento del programa, sino como entidad existente en tiempo de ejecución.

Sin embargo, esta definición tan estricta hace que algunas de tales aproximaciones, sean muy criticadas en su forma de utilizar la Orientación a objetos.

Un ejemplo es el Sistema Operativo Choices[CIM92], que únicamente aplica la OO en el diseño del sistema.

La infraestructura para este mundo de objetos debe proporcionar mecanismos para[GKS94]:

- **Existencia**. El sistema operativo debe determinar una representación interna que represente la vida del objeto desde su creación hasta su destrucción, de tal forma que sea posible crear instancias de objetos e incluso proporcionar nuevas descripciones de objetos.
- **Identificación**. El sistema operativo debe identificar al objeto mediante algún identificador interno que permita que otros objetos lo conozcan con el fin de permitir la interacción entre ellos. A la vez, debe proporcionar mecanismos para localizar el objeto.
- **Interacción**. Es necesario proporcionar algún mecanismo para que los objetos puedan interaccionar, no sólo entre instancias de usuario sino también con objetos del sistema, de una manera uniforme.

Sistema Operativo como Conjunto de Objetos

Además de ofrecer soporte y dotar de entidad propia a la abstracción de objeto, el propio sistema operativo debe estar construido en función del propio concepto de objeto.

Así pues, el sistema operativo estará formado a su vez por un conjunto de objetos, algunos de los cuales serán visibles explícitamente a los objetos de usuario proporcionándoles la funcionalidad del sistema operativo que necesiten y formando la interfaz OO del sistema operativo a los objetos de usuario. El acceso a esta interfaz se llevará a cabo igual que el acceso a cualquier otro objeto, mediante el mecanismo de comunicación que se defina.

Otros, no serán visibles al conjunto de objetos de usuario, aunque serán esenciales para la existencia y la interacción de instancias.

Entorno Flexible: Un Requisito Más

Si el soporte a objetos y la organización como objetos del SOOO son los dos requisitos más reseñables de la construcción de un SOOO, es necesario destacar también un tercero, la construcción de un entorno flexible, como un requisito tan importante como los anteriores, que nos permita personalizarlo y salir del tradicional corsé que supone el Sistema Operativo para las aplicaciones.

2 Soporte al concepto de objeto

Uno de los principales requerimientos de diseño, sino el más, es ofrecer una abstracción de alto nivel a los usuarios, de forma que pueda ser visto como una máquina virtual y pueda ser considerado como un entorno de programación.

La cuestión que se plantea es cuántas y qué tipo de abstracciones definir, además de determinar cuáles son las propiedades de tal abstracción.

En un SOOO es requisito imprescindible proporcionar soporte a los objetos. Este soporte debe ser genérico, sin casarse con ningún modelo de objetos de un lenguaje de programación particular.

A continuación se detallan las cuestiones de diseño que se deben tener en cuenta.

2.1 Uniformidad en torno a la OO

Los sistemas operativos tradicionales proporcionan múltiples visiones distintas a los usuarios. Por ejemplo, en Unix[Bac86] hay dos abstracciones fundamentales en el sistema: los ficheros y los procesos. Los ficheros son entidades abstractas y estáticas que representan los recursos gestionados por el sistema. Por su parte, los procesos son entidades dinámicas que acceden y manipulan los ficheros.

2.1.1 Abstracción única

La propuesta que se defiende en la presente tesis es que el sistema integral ofrezca una única abstracción heredable por las capas superiores del mismo. Esta abstracción es el objeto, sea cual sea su granularidad. Esto mismo se propone en otros sistemas OO como Muse[YTM+91], antecesor de Apertos[Yok92].

Tanto el Sistema Operativo como las aplicaciones de usuario se estructurarán, en tiempo de ejecución, como objetos que interaccionan entre sí y llevan a cabo operaciones ante la invocación de sus métodos, independientemente, de que el lenguaje utilizado para definir la aplicación pueda ser o no OO.

2.2 Modelo de Objetos único intencionadamente estándar

El sistema operativo, como parte del sistema integral descrito en el Capítulo V, se organiza en torno al paradigma de OO dando soporte a un modelo de objetos, es decir, definiendo un conjunto de características para los objetos.

Así, un objeto deja de ser una simple zona de memoria que el sistema operativo maneja de forma conjunta para convertirse en la unidad de construcción de sistemas en

tiempo de ejecución dotado de una comportamiento y una semántica completas[ATA+98, Yok92].

El sistema operativo propuesto define un **modelo único de objetos**, es decir, todos los objetos se definen siguiendo las mismas propiedades esenciales aunque pueden tener un comportamiento distinto[ATA+98].

Se buscó de forma intencionada que este conjunto de propiedades esenciales fuesen las más comunes y ampliamente aceptadas en los lenguajes de programación, encapsulación, herencia y polimorfismo[Boo91], buscando un acercamiento y una mayor facilidad a la hora de dar soporte a los modelos de los lenguajes.

Además de estas propiedades, algunas otras más avanzadas como la persistencia, la concurrencia o la distribución, también son ampliamente aceptadas como propiedades del modelo de objetos.

En los capítulos VII, VIII y IX se definen las características del modelo de objetos para la concurrencia.

Algunos sistemas como COOL[HMA90] o Tigger[Cah96b] no ofrecen un soporte único para objetos sino que están enfocados a poder soportar distintos modelos de objetos en tiempo de ejecución. Para lograrlo, ofrecen un soporte genérico al que se añade el soporte necesario para cada lenguaje.

La principal ventaja es que son capaces de soportar varios modelos de forma directa y se adapta completamente al lenguaje. Sin embargo, tiene dos inconvenientes muy serios, la complejidad añadida y la falta de uniformidad.

2.3 Abstracción Homogénea

La única abstracción ofrecida por el sistema integral y heredable por las capas superiores del mismo es el objeto, sea cual sea su granularidad[YMF+91].

Granularidad de los Objetos

Entre los sistemas orientados a objeto, hay diversidad de definiciones del concepto de objeto basándose en la granularidad del mismo[CC91, YMF+91].

- **Objeto = Objeto Grano grueso**. Hay sistemas operativos OO que ofrecen únicamente objetos de grano grueso, de tal forma que sólo aquellas entidades que implican grandes cantidades de datos se representan como objetos. Sin embargo, los datos pequeños como enteros o caracteres, se tratan de forma ordinaria y no se consideran objetos. La justificación que ofrecen se basa en el coste de crear y gestionar entidades pesadas como son sus objetos[Ame88], aunque otros trabajos como el lenguaje Smalltalk-80[GR83] muestran que es factible representar cada dato como un objeto, hecho este que ofrece una visión uniforme de todas las partes del sistema. Ejemplos de estos sistemas son Eden[ABL+85], Clouds[DLA+91], Chorus[RAA+88], Amoeba[TRS+90] o Java[Sun97a, GJS96]. Sin embargo, la tendencia es a definir objetos de cualquier tamaño, por lo que se hace difícil integrar la definición de objetos en este sistema y, por tanto, los objetos del lenguaje son diferentes de los objetos del sistema.
- **Objeto = Objeto de Grano Fino**. En el lado contrario, Emerald[BHJ+86, Hut87] y Orca[BKT89], por citar algunos, ofrecen objetos de grano fino. En este tipo de

sistemas los mecanismos de creación y destrucción, así como la comunicación entre objetos son cruciales.

Objetos Homogéneos

En el entorno de objetos propuesto, todos los objetos del sistema tienen la misma consideración, no hay objetos especiales que reciban un trato distinto, estén dotados de características especiales. De tal forma que los propios objetos del sistema operativo no son diferentes del resto de los objetos de usuario[YTT89, YMF+91]. En general, todos los objetos tienen la misma semántica[Hau93].

Para lograr construir un sistema homogéneo, la máquina abstracta, como corazón del SIOO, ofrece la Clase Objeto como raíz. Cualquier objeto será una instancia de esta clase. Los objetos muy pequeños o básicos, como los enteros, se ofrecen como objetos predefinidos, pero no como tipos.

Esta simplicidad conceptual hace que todo el sistema en su conjunto sea fácil de entender, y se elimine la desadaptación de impedancias al trabajar con un único paradigma. La tradicional distinción entre los diferentes elementos de un sistema: hardware, sistema operativo y aplicaciones de usuario se difumina, sin necesidad de perder la eficiencia ya que, la propia implantación de la máquina puede hacerse de forma que se traten distinto los enteros que los ficheros, por ejemplo.

2.3.1 Modo de trabajo exclusivamente orientado a objetos

La única abstracción es, por tanto, el objeto (de cualquier granularidad), que encapsula toda su semántica. Lo único que puede hacer un objeto es crear nuevas clases que hereden de otras, crear objetos de una clase y enviar mensajes[1] a otros objetos. Toda la semántica de un objeto se encuentra encapsulada dentro del mismo.

3 Estructuración e implantación del sistema como un conjunto de objetos

A la hora de construir un SOOO según la definición dada al principio del Capítulo, es necesario definir la propia estructura del Sistema Operativo de forma que encaje con el paradigma de OO.

3.1 Diseño del Sistema Operativo como un marco OO

Una primera aproximación en la introducción del paradigma de OO en los sistemas operativos es aplicar los principios de diseño e implantación al propio sistema operativo de forma que este quede organizado como un marco OO, diseñándolo como un conjunto de jerarquías de clases cada una de las cuales ofrece la funcionalidad de las distintas partes del sistema operativo.

El diseñar un sistema orientado a objetos, sea el sistema operativo u otro, tiene múltiples ventajas ampliamente descritas en [Rus91b, Cah96a] y que se describen también tanto en el Capítulo II como más en detalle al final de este capítulo.

[1] Se utilizarán indistintamente las expresiones envío de mensaje, llamada o invocación de método y llamada o invocación de operación.

Tales ventajas, como son la reutilización del código, o la portabilidad, en el caso del sistema operativo cobran, si cabe, mayor importancia. El ejemplo más clásico de este tipo de sistemas es Choices [CIM+93].

Aunque el sistema pueda estar estructurado y se implemente como un conjunto de objetos, es posible que no dé soporte a objetos definidos desde el exterior y que siga ofreciendo a los usuarios y aplicaciones unos servicios tradicionales y una interfaz también tradicional para usar los servicios de sistema que ofrece. De ello nos ocupamos a continuación.

3.2 Sistema Operativo como conjunto de objetos en tiempo de ejecución

El Sistema Operativo se estructura internamente como un conjunto de objetos que implementan la funcionalidad del sistema. Pero no sólo es una estructuración en tiempo de diseño sino que, en tiempo de arranque y en tiempo de ejecución, se instanciarán objetos reales de las distintas clases en que se oferta la funcionalidad del sistema operativo[Camp91] con la semántica que ello conlleva.

3.3 Interfaz oo

Dado que el propio sistema operativo se constituye como un conjunto de objetos en tiempo de ejecución, ofrecerá la funcionalidad a través de los métodos de los objetos.

De esta forma, la interfaz del sistema operativo no se distingue de la que ofrece cualquier otra aplicación.

4 Multilingual

El sistema operativo debe ser capaz de soportar los requisitos de múltiples lenguajes de programación dando soporte a un único modelo de objetos independiente del lenguaje.

Objetos del Lenguaje versus Objetos del Sistema

Muchos SOOO distinguen entre los objetos del lenguaje y los objetos del sistema, siendo los primeros los que representan a los objetos definidos por las aplicaciones y los segundos, los que representan a los objetos que encapsulan la funcionalidad del sistema operativo. Es decir, definen dos tipos de objetos que se diferencian en alguna propiedad.

Sin embargo, esta aproximación ofrece varias desventajas[YMF+91].

- **Violación del Principio de Economía de Conceptos**. Los programadores deben estar pendientes de dos tipos de objetos a la hora de programar.
- **Déficit para los objetos del Sistema**. Aunque el sistema en tiempo de ejecución de los lenguajes ofrecen un rico soporte de programación, de esto sólo se benefician los objetos del lenguaje, los objetos del sistema no pueden beneficiarse de esto. Por ejemplo, los lenguajes OO proporcionan una jerarquía de clases. Sin embargo, es difícil reutilizar los objetos ya existentes debido a que tales objetos no pueden ser definidos en tal jerarquía.

- **Déficit para los Objetos del Lenguaje**. Los objetos del lenguaje no pueden beneficiarse de los servicios proporcionados por los objetos del sistema. Por ejemplo, en un sistema que proporciona migración de objetos, los objetos del lenguaje no podrían beneficiarse de la migración de objetos, dado que el sistema no tiene forma de saber el estado del lenguaje.

5 Flexibilidad

5.1 Flexibilidad: Nuevo Reto en el Diseño de Sistemas Operativos

La **flexibilidad** se define como la habilidad de diseñar un sistema que ofrezca la posibilidad de ser personalizado para cumplir los requisitos específicos de aplicaciones o dominios de aplicación[Cah96a].

Diseñar un sistema operativo flexible supone que sea fácilmente personalizable para proporcionar a cada aplicación aquellas características que requiera, es decir, permitir a cada aplicación o conjunto de aplicaciones, configurar el Sistema Operativo de la forma más adecuada. De hecho, dado que algunas aplicaciones pueden funcionar de forma poco eficiente con la implantación concreta que el sistema operativo ofrezca por defecto, la posibilidad de personalizar una característica trae consigo el aumento del rendimiento.

5.2 Falta de flexibilidad. Causas y Consecuencias

5.2.1 Las Causas

Existen una serie de factores que determinan el hecho que un sistema operativo sea inflexible y varios autores como [Dra93] han realizado trabajos en los que se identifican esas causas y se muestra cómo su solución contribuye a un Sistema Operativo más flexible. En [Cah96a] se presenta un interesante trabajo comparativo del grado de flexibilidad en distintos sistemas operativos actuales.

Problemas para la Flexibilidad: Ocultación de información

El origen de todos los problemas estriba en el comportamiento tradicional de los Sistemas Operativos. Estos han constituido siempre auténticas **cajas negras** que elevan el nivel de abstracción ocultando los detalles de implantación y con ello, las decisiones tomadas para una determinada característica[KLM+93].

De aquí se derivan el resto de los problemas que se enuncian a continuación.

Abstracciones y mecanismos generales

Los sistemas operativos tradicionales, normalmente, proporcionaban gran cantidad de abstracciones y mecanismos de propósito general, con el fin de adaptarse a los requisitos de la mayor cantidad posible de aplicaciones. Dicho de forma rápida, proporcionaban una plataforma lo menos mala posible para la mayoría de las aplicaciones.

Sin embargo, esto resulta, en muchos casos, contradictorio. Habrá gran cantidad de aplicaciones que no necesiten muchas de las características ofrecidas. Es más, la existencia de las mismas le provoca una sobrecarga que empeora su rendimiento.

Además, estas abstracciones y mecanismos proporcionados por el sistema operativo son fijos e inamovibles y de ello se derivan una serie de problemas propios de la adaptación de aplicaciones diversas con sistemas operativos tradicionales.

Deficiencia de características

Otra causa se presenta cuando el sistema operativo no proporciona alguna característica requerida por la aplicación. Puede ser posible modificar el sistema operativo directamente como en Delta[CPR+92] para tolerancia a fallos o Grashopper[Lin95] para persistencia.

Sin embargo, esta aproximación es altamente inflexible ya que elimina cualquier posibilidad de personalización.

Otra posibilidad es proporcionar la funcionalidad requerida en forma de librerías. Aunque esta aproximación sí presenta la ventaja de ser personalizable por el usuario, también arrastra el inconveniente de que no es transparente al código funcional de la aplicación.

Versión inadecuada

Por último, está el problema de la versión inadecuada, que se presenta cuando la aplicación necesita, para su correcto funcionamiento, una versión diferente del sistema operativo o de partes del mismo, a la que está utilizando.

5.2.2 Las Consecuencias

Rendimiento pobre

Debido a que el sistema operativo ofrece servicios implementados de forma genérica, es más que probable que tales servicios no estén implementados de forma óptima para todas las aplicaciones. Es más, la forma concreta e inamovible en que el Sistema Operativo implementa sus servicios, constituye una desventaja para un conjunto más o menos amplio de aplicaciones, provocando que estas no alcancen un rendimiento óptimo.

Imposibilidad de adaptar

Debido al comportamiento de caja negra de los sistemas operativos que ya se mencionó en el Capítulo II y anteriormente en este mismo Capítulo, estos proporcionan implantaciones particulares de sus servicios, hechas para soportar los requisitos de las aplicaciones más genéricas, y que ocultan a los usuarios las decisiones de implantación tomadas.

Puede suceder que las decisiones de implantación tomadas no sean las más adecuadas para una aplicación concreta, produciéndose entonces un *mapping conflict* (Véase Capítulo II).

Estos sistemas operativos, además de ocultar estas decisiones, no ofrecen ningún mecanismo para modificarlas o personalizarlas.

Una aproximación para solucionar el problema es explotar el conocimiento que las aplicaciones tienen acerca de cómo implementar las distintas características del sistema operativo. Esto proporcionaría una gran flexibilidad a la hora de construir el sistema operativo y una considerable mejora en el rendimiento.

Igualmente, se desea dotas al sistema integral de un diseño que permita la extensibilidad, es decir, que pueda ir añadiendo modificaciones que garanticen la integridad del sistema operativo ante cambios del sistema. La presencia de esta característica permite a un sistema operativo ser fácilmente mejorado.

5.3 Distintas aproximaciones a la flexibilidad

Básicamente, existen dos aproximaciones para conseguir un sistema flexible: **flexibilidad estática y dinámica**[Cah96a].

5.3.1 Flexibilidad estática: Sistemas personalizables

La flexibilidad estática permite que el sistema sea personalizado para una aplicación, conjunto de aplicaciones o bien para alguna configuración hardware particular **en tiempo de compilación, enlace o carga**.

Para lograr un sistema estáticamente flexible, se requiere poder implementar de diversas formas la arquitectura del sistema con el fin de incluir los mecanismos y las políticas requeridos por las aplicaciones.

La arquitectura de un sistema está compuesta por distintos subsistemas de los cuales pueden existir distintas implantaciones. Para construir un sistema flexible en este sentido, es preciso poder elegir en tiempo de compilación, enlace o carga una implantación concreta de entre las existentes, o bien proporcionar una nueva.

Como ejemplos de tales sistemas están Choices[CIM92] y PEACE[SPr93b].

Se dice además que estos sistemas son extensibles o están dotados de una arquitectura extensible, si se han diseñado para facilitar la introducción de nuevos subsistemas que proporcionan funcionalidad adicional que no está cubierta todavía por la arquitectura.

5.3.2 Flexibilidad dinámica

Es un paso más a la hora de lograr un sistema flexible y supone que el sistema puede ser personalizado para cubrir o adaptarse a las necesidades de las aplicaciones **en tiempo de ejecución**. Para conseguir la flexibilidad dinámica existen diversos grados que se detallan a continuación.

Sistemas Adaptables

Una de las principales causas de la desadaptación entre los sistemas operativos tradicionales y las aplicaciones actuales viene derivada de un problema intrínseco al propio diseño de los sistemas operativos.

No ya los mecanismos, sino las políticas del sistema operativo están incrustadas en el núcleo del sistema evitando de esta forma la evolución del sistema para adecuarse a los requerimientos de las aplicaciones[GIL95].

Un sistema adaptable se construye como un sistema de propósito general, posiblemente basado en las mismas abstracciones y mecanismos, pero proporciona servicios que soportan un amplio rango de políticas que dan cobertura a los requisitos del mayor número posible de aplicaciones

Sin embargo, su principal característica es que permite la personalización para cambiar dinámicamente durante la ejecución del programa de tal forma que pueda adecuarse dinámicamente a los requerimientos de las aplicaciones.

Para ello, además de proporcionar servicios que cubren un amplio rango de requisitos, ofrece las interfaces que permitan a las aplicaciones determinar la política requerida de acuerdo a sus restricciones y necesidades.

Un ejemplo de tal tipo de sistemas es el micronúcleo Mach que permite a las aplicaciones dar pistas al sistema de planificación[Bla90].

Si el sistema por sí mismo fuese capaz de elegir la política apropiada para utilizar para cada tipo de aplicaciones basándose en parámetros como un histórico de comportamiento o similares, tal sistema se cataloga como **adaptativo**.

Sistemas Modificables

Un paso más es permitir a la aplicación, no ya escoger entre un amplio rango de políticas ofrecidas, sino participar de alguna forma en la implantación de un servicio. Esta aproximación se consigue obligando al servicio del núcleo a efectuar una upcall a un módulo proporcionado por la aplicación que implementa alguna política para el servicio. Otra posibilidad es permitir que la aplicación interponga código a la interfaz del servicio.

Un ejemplo de un sistema que soporta esta clase de flexibilidad es el micronúcleo Mach vía la interfaz con el paginador externo[YTR+87].

Sistemas Configurables y Extensibles

La tercera aproximación para conseguir sistemas dinámicamente flexibles es construir un sistema que proporcione algún medio que permita a las aplicaciones reemplazar en tiempo de ejecución servicios del sistema para dar soporte a funcionalidad de diferente modo, para una aplicación particular. Estos sistemas reciben el nombre de **configurables**.

Si además permiten añadir nuevos servicios en tiempo de ejecución con el fin de proporcionar funcionalidad nueva, tales sistemas reciben el nombre de **extensibles**. Algunos trabajos proponen que la extensibilidad puede conseguirse diseñando el núcleo del sistema operativo como un conjunto de componentes que las aplicaciones pueden reutilizar para dotar al sistema de nueva funcionalidad[SS95].

La mayoría de los micronúcleos son configurables dado que por su propia arquitectura tienen muchos módulos del sistema operativo implementados como servidores en el espacio del usuario por lo que pueden ser instalados o reemplazados en tiempo de ejecución. Sin embargo, tienen el inconveniente de que crear nuevos componentes y reemplazar los ya existentes es complicado y supone una considerable sobrecarga ya que los servidores son normalmente grandes programas.

Otro inconveniente es que los propios servidores, a pesar de estar en el espacio de usuario, están tan protegidos de las modificaciones de las aplicaciones de usuario como si estuvieran en el kernel al ejecutarse en espacios de direcciones separados.

Otra posibilidad de extensión es enlazar extensiones directamente en el kernel. SPIN va más allá y permite incluso instalar servicios dinámicamente en el kernel. Aunque mejora el rendimiento, pero es necesario añadir mecanismos que prevengan la corrupción de los componentes ya existentes y que permitan el borrado de código biejo cuando ya no se necesita.

5.4 Flexibilidad en los Sistemas Operativos

Existen varias aproximaciones para solucionar los problemas derivados de la ocultación de la información como la deficiencia de características o la versión inadecuada de alguna de ellas y dotar al Sistema Operativo de flexibilidad [KG96, HPM+97, LYI95]:

- **Solución ad-hoc**. En el caso de una deficiencia de características o una característica poco adecuada a la aplicación que se ejecuta, es posible **extender o modificar el sistema operativo directamente** como en Delta[CPR+92] para añadir tolerancia a fallos o Grashopper[Lin95] para dotar de persistencia al sistema. Sin embargo, esta aproximación es altamente inflexible ya que elimina cualquier posibilidad de personalización y tampoco permite la extensión del sistema.
- **Extender el Sistema Operativo por medio de Librerías**. Otra posibilidad es proporcionar la funcionalidad requerida en forma de librerías. Esta aproximación tiene la ventaja de ser personalizable por el usuario, pero no es transparente al código funcional de la aplicación, además de llevar intrínsecamente ligadas algunas incomodidades como la necesidad de algún código extra. Por ejemplo, la introducción del modelo asíncrono y de la comunicación remota en Eiffel, fuerza a emplear cierta cantidad de código de manipulación explícita de mensajes[KB93], en oposición al paso de mensajes estándar implícito.
- **Flexibilidad a través de la Modularidad**. La modularidad es una técnica clave a la hora de construir software flexible dado que encapsula los componentes en módulos independientes y facilita su cambio y evolución. Esta técnica, popularizada por los micronúcleos, hace que estos ganen cierta flexibilidad al descomponer los sistemas monolíticos en núcleo y servidores a nivel usuario lo que permite a los programadores reemplazar componentes a nivel usuario y configurar el sistema a medida. Sin embargo, la creación y modificación de estos componentes es una tarea complicada y la penalización en el rendimiento suele ser alta[CL95]. La construcción de sistemas operativos orientados a objeto ha extendido esta modularidad con un grado de granularidad mucho más fino y, con el uso de técnicas como la especialización a través de la herencia permiten soportar un alto grado de flexibilidad[LJP93].
- **Adaptación en tiempo de Ejecución**. Una aproximación más es permitir enlazar extensiones directamente en el núcleo, como SPIN[BCE+94] o los exonúcleos como Aegis[EKT95]. Si bien puede ofrecer un buen rendimiento, se requiere un esfuerzo extra para evitar que el nuevo código enlazado con el núcleo pueda corromper los componentes existentes, así como para eliminar el código obsoleto cuando ya no es necesario por más tiempo.
- **Abrir la Caja: Reflectividad**. Se propondrá dotar al sistema de una arquitectura reflectiva para solucionar este y otros problemas. Esta es una solución muy

prometedora como se verá en el Capítulo XII y como proponen autores como[Foo90, GW96, LYI95].

6 Reflectividad. Arquitectura para un sistema flexible

6.1 OO: Necesaria, pero no suficiente

6.1.1 Necesaria ...

Es innegable que la construcción de un Sistema Operativo aplicando Tecnologías OO, deriva en la implantación de sistemas más modulares, incrementando la reutilización y facilitando el mantenimiento de los mismos, a la vez que da a los usuarios o programadores una perspectiva única del sistema alrededor del concepto de objeto. Esto facilita enormemente construir sistemas grandes y complejos, como un Sistema Operativo.

6.1.2 ... Pero no Suficiente

En un marco como el que se define en este Sistema Operativo, el objeto es el único elemento constituyente en el sistema. La organización en torno al concepto de objeto hace al sistema más abierto y con mayores posibilidades de evolución.

Sin embargo, a pesar de esto, la ejecución de los objetos sigue estando restringida por el entorno de ejecución que se defina. Por ejemplo, es posible definir algunos objetos en el mundo real que tengan límites de tiempo para realizar sus tareas o que deban ser más estables que otros.

No podemos dar al objeto un conjunto único de propiedades y una semántica fija y predeterminada, ya que, tanto las propiedades como la semántica de los objetos cambian a medida que el objeto se ejecuta, a medida que evoluciona. Por tanto, las características de cada objeto deben acomodarse de forma personalizada a cada uno de ellos, de tal forma que se consiga un mundo homogéneo de objetos heterogéneos.

La reflectividad, propuesta inicialmente en el campo de los lenguajes de programación para extender la semántica de estos dinámicamente[Smi84], se está extendiendo a otros campos, como son los sistemas operativos[LYI95].

En el contexto de los Sistemas Operativos, dotarlos de una arquitectura reflectiva, supone que el sistema será capaz de proporcionar una separación clara entre el modelo computacional ofrecido al usuario y el meta-modelo computacional, la forma en que la computación de usuario se lleva a cabo[TNO92].

Dado que este es un eje central en esta tesis, se realizará un tratamiento en profundidad en el Capítulo XII. Baste aquí con introducir la idea fundamental.

6.2 Separación Objeto – Meta-objeto

Para lograr un entorno adaptable para los objetos, en esta tesis se propone dotar al sistema de una arquitectura reflectiva, separando objetos y meta-objetos en el diseño del

sistema Operativo. Un objeto será un contenedor de información, mientras un meta-objeto define la semántica de su comportamiento. Cada objeto tendrá su grupo de meta-objetos que ofrece al objeto metaoperaciones que definen su semántica y puede acceder a la representación interna del objeto.

Estos meta-objetos proporcionan una interfaz transparente gracias a la cual los objetos acceden a los servicios del sistema operativo necesario para soportar un requerimiento particular de una aplicación determinada, siendo además posible personalizar la implementación de tales meta-objetos[WS95].

Así, se pueden cubrir nuevos requerimientos particulares de una aplicación implantando y proporcionando al sistema nuevos meta-objetos sin que sea necesario reimplementar o hacer ningún cambio en los objetos de la aplicación.

La arquitectura reflectiva proporciona al menos tres primitivas. La primera permite a un objeto hacer una petición de meta-computación. La segunda dota a los meta-objetos de la habilidad de reflejar el resultado de la meta-computación en los objetos de nivel base y la tercera, mantiene la conexión causal entre objetos y meta-objetos.

6.2.1 Meta-Objetos: Objetos

Siguiendo el principio de uniformidad en torno al concepto de objeto, los meta-objetos del sistema se representarán como objetos, de tal forma que responderán a los mensajes de su interfaz, encapsularán sus datos, etc.

Al ser objetos, a su vez tendrían que tener una plataforma formada por meta-objetos. Esta recursividad se termina al ser la máquina la plataforma de soporte de estos.

7 Ventajas derivadas de un Sistema Operativo OO

La aplicación de las técnicas de Orientación a Objeto al diseño e implantación de los sistemas operativos, proporciona una serie de ventajas comunes a todo sistema software que se construya siguiendo dicho paradigma, como ya se vio en el Capítulo II.

Sin embargo, tan importantes o más que estas, son las ventajas que se logran de la aplicación de los principios de OO al soporte en tiempo de ejecución de aplicaciones y del propio sistema Operativo.

7.1 Economía de Conceptos

La uniformidad del Sistema operativo en torno a la OO, ofreciendo el objeto como única abstracción, así como la homogeneidad del modelo soportado, integrando en el concepto de objeto no sólo las propiedades básicas, sino también la concurrencia, la persistencia y la distribución, proporcionan un modelo de programación basado en un conjunto mínimo de conceptos y un marco único para los programadores.

Esta economía de conceptos deriva en un mejor aprendizaje, un factor de incidencia mayor y un rendimiento mayor[BG96].

7.2 Rendimiento

El marco uniforme proporcionado por la suma de máquina abstracta más sistema operativo, ambos construidos en torno al concepto de objeto, conlleva que las aplicaciones que se ejecutan sobre este último, vean una plataforma de programación y ejecución adecuada que habla el mismo idioma que ellos: conoce y gestiona las mismas abstracciones y ofrece sus servicios de la misma forma que cualquier otra aplicación.

De esta forma, la integración, no ya de distintas aplicaciones OO sino de aquellas basadas en distintos modelos de objetos, se consigue de forma directa.

Gracias a esto, se eliminan las capas intermedias que se habían dado como solución transitoria en el Capítulo I, con la consiguiente mejora del rendimiento.

7.3 Sistemas Personalizables

En los sistemas operativos orientados a objetos, la funcionalidad del sistema operativo se diseña como un conjunto de clases que implementan el comportamiento de los distintos componentes del núcleo.

Es posible especializar estas clases para adaptarse a los requerimientos de una determinada aplicación o aumentar la funcionalidad de las clases. Incluso es posible reemplazar un componente por otro.

Con esto se consigue modificar la forma habitual de implementar el Sistema Operativo, en la que las decisiones se dejan en manos del implementador del sistema, que elige la opción menos mala para la mayoría de las aplicaciones. En los sistemas personalizables se permite que cada aplicación seleccionase su propia política y el papel del sistema operativo se reduciría a utilizar una u otra y a mediar entre ellas[SS95], lo que, evidentemente es una solución mejor.

La modularidad impuesta por el diseño orientado a objetos, la herencia y el polimorfismo, son los elementos clave para lograr el desarrollo de un sistema operativo personalizable.

7.4 Adaptabilidad

La adaptabilidad de un sistema operativo permite la personalización del mismo para cambiar dinámicamente durante la ejecución del programa de tal forma que pueda adecuarse dinámicamente a los requerimientos de las aplicaciones.

Un Sistema Operativo Orientado a Objeto estructurado él mismo en torno al concepto de objeto, ofrece la funcionalidad del Sistema Operativo como objetos que responden a mensajes de una interfaz o invocación a método. Esto favorece la adaptabilidad de grano fino al encapsular toda la funcionalidad en objetos.

El conjunto de objetos del sistema operativo instanciados para cualquier aplicación, forma la plataforma o entorno de ejecución de tal aplicación. Estos objetos se representarán como objetos normales del sistema, siguiendo el principio de uniformidad.

El hecho de que estos objetos sean igual que todos los demás, instancias de clases que pueden ser creados en tiempo de ejecución, permite a las aplicaciones modificarlos o sustituirlos, de tal forma que los mecanismos y políticas que tradicionalmente están

fijos como parte del sistema y que ahora están contenidos en los objetos del sistema operativo, puedan ser modificados.

7.5 Extensibilidad: Soporte para interfaces adaptables

La extensibilidad de un sistema operativo permite a las aplicaciones, no sólo modificar la funcionalidad del sistema operativo, sino además, permite a las aplicaciones incorporar nueva funcionalidad al sistema en ejecución, evitando la necesidad de recompilarlo y rearrancarlo para acomodarse a las necesidades imprevistas.

Los modelos de extensibilidad actuales (por ejemplo, el paginador externo de Mach) ofrecen únicamente extensibilidad de grano grueso.

Una forma de extender un sistema operativo es añadiendo nuevos componentes a la interfaz de la aplicación[Rus91].

Dado que el Sistema Operativo propuesto se compone de objetos, la interfaz de aplicación de este sistema se define por el conjunto de objetos que se ponen a disposición de las aplicaciones. Este conjunto de objetos puede cambiar dinámicamente según las necesidades de la aplicación.

Estos objetos son instancias de clases del sistema que el usuario puede especializar o incluso crear las suyas nuevas, aun cuando la aplicación ya se esté ejecutando.

De esta forma, extender el sistema operativo supone crear nuevas instancias de estas nuevas clases en tiempo de ejecución. Estos objetos responderán a los mensajes que les envíen las aplicaciones. Únicamente es necesario informar a estas del identificador del objeto para enviarle mensajes.

En el caso de que el sistema operativo esté basado en un servidor de nombres que proporciona a los objetos de aplicación, las referencias a los objetos servidores necesarios, esto puede lograrse modificando, añadiendo o borrando los enlaces entre objetos y nombres en el servidor de nombres de la aplicación. Es posible asignar diferentes servidores de nombres a diferentes aplicaciones permitiendo que tengan diferentes interfaces al sistema operativo.

Capítulo VII

Diseño de un Entorno de Computación para el Sistema Operativo OO

Tal y como se mostró en el Capítulo III, los modelos de computación tradicionales, basados en la abstracción de proceso o tarea, no encajan demasiado bien con los principios de OO, *alma matter* de esta tesis.

Se hace por tanto necesario buscar una nueva forma de integrar computación y objetos, que permita introducir la concurrencia en el sistema, integrándola con la abstracción de objeto para mantener la uniformidad.

Sin embargo, algo como introducir la concurrencia en el sistema operativo, que semeja una tarea sencilla, se verá que no lo es tanto, hasta el punto de hacer peligrar propiedades tan elementales como la encapsulación. Por tanto, habrá que poner especial cuidado en el modelo que se elige, garantizando la homogeneidad y la uniformidad.

Este capítulo, prólogo de los tres que siguen a continuación, se estructura en dos partes, como sigue. En la primera, se hace un breve repaso a los objetivos más importantes en el diseño del sistema de computación, que serán tratados más a fondo en los capítulos VIII y IX. En la segunda, se presentan las tres aproximaciones para la integración de computación en un entorno OO, que a su vez serán tratadas más profundamente en el capítulo X.

1 Objetivos de diseño del sistema de computación para el SOOO

El objetivo fundamental a la hora de diseñar el sistema de computación del SOOO, es integrar el concepto de objeto y la computación del mismo, manteniéndonos fieles a los principios de uniformidad y homogeneidad. Este objetivo fundamental no es más que una extensión del objetivo ya visto en el capítulo VI de **abstracción única**, que se mencionará aquí nuevamente, refiriéndose en este caso a la integración de la computación de forma natural con el concepto de objeto, sin añadir nuevas abstracciones que compliquen la vida al programador (**Economía de Conceptos**) [KB93, Nie93, VBJ+96].

1.1 Requerimientos funcionales

Abstracciones adecuadas

Visto el fracaso de la integración del paradigma de OO con los entornos tradicionales del Capítulo III, la decisión de en qué forma se integra la concurrencia en el modelo, garantizando en todo caso la integridad del objeto, debe ser tomada cuidadosamente.

Algunos de los modelos tradicionales vistos en el Capítulo III adolecen del problema de que las abstracciones que ofrecen no son adecuadas para su integración con el paradigma de la OO por varios motivos, fundamentalmente, por ofrecer una abstracción distinta al objeto.

La aproximación del modelo de objeto/hilo, aunque prometedora, supuso una decepción ya que no supone la integración de dos conceptos como el objeto y la tarea o actividad, sino que, estructura el entorno alrededor de dos entidades distintas, el objeto, entidad pasiva y el hilo, entidad activa, con semántica y comportamiento diferentes.

El sistema Clouds[DLA+91] es un ejemplo de sistema que da soporte a objetos distintos mientras que Apertos[Yok92] y su antecesor Muse[YTT89] son ejemplos de sistemas que proponen la encapsulación de la computación y los datos en una única abstracción.

Más tarde en este mismo capítulo se realizará un estudio de las distintas aproximaciones, aunque se puede avanzar aquí que en esta tesis se propone seguir un camino similar a Muse y Apertos, para lo que será necesario dotar a la entidad de objeto de una semántica mucho más potente que ser una zona de memoria. Es decir, de qué propiedades dotará el modelo de objetos a nuestra abstracción de objeto.

Comunicación y sincronización

La interacción entre objetos tendrá lugar cuando un objeto envíe un mensaje a otro (invocación de métodos) y existen distintos modelos de comunicación, que se verán en el capítulo VIII.

Independientemente del modelo de comunicación elegido, la única comunicación posible entre objetos, será pues, vía paso de mensajes (**modo de trabajo exclusivamente oo**).

Debido a esto, el envío de mensajes supone, no sólo la comunicación entre objetos, sino también un mecanismo útil y válido para transferir el control entre objetos. De esta forma, la invocación de métodos entre objetos puede suponer un mecanismo de sincronización para la ejecución de actividades entre objetos y, paradójicamente, puede implicar la necesidad de un mecanismo de sincronización interno al propio objeto que le permita proteger la consistencia de su estado interno ante múltiples invocaciones[CC91, Nie93].

Entorno de Computación Concurrente

El entorno de computación no debe limitar innecesariamente el grado de concurrencia o métodos a ejecutar, antes bien, debe ofrecer la máxima libertad posible, así como los mecanismos para limitarla en caso necesario.

De tal forma que, no sólo se puedan ejecutar varios métodos de distintos objetos (**Concurrencia entre objetos**), sino varios métodos de un mismo objeto (**Concurrencia Intra-Objeto**).

Modelos como Apertos[Yok92] o Muse[YTT89] definen un objeto que secuencializa sus métodos lo que no sólo proporciona una abstracción más sencilla, sino también un mecanismo efectivo y robusto para el control de la concurrencia a cambio de limitar el grado de concurrencia[MHM+95].

Planificación

Un problema clásico en los Sistemas Operativos es el relativo a la planificación de las actividades que se llevan a cabo concurrentemente. Dado que, no sólo pueden existir actividades ejecutando métodos de distintos objetos, sino incluso métodos de un mismo objeto, el sistema operativo debe ocuparse, tanto de la planificación interna como de la externa.

1.2 Requerimientos no funcionales

Los requisitos no funcionales establecen los parámetros a los que el sistema debe ajustarse a la hora de cumplir los requisitos funcionales.

El conjunto de requerimientos no funcionales propuesto para el sistema operativo expuesto en esta tesis, coinciden en gran parte con los que se aplican tradicionalmente a todo sistema operativo[Cro96] y se verán con más detalle en el capítulo IX.

Abstracción única y sencilla

Las abstracciones ofrecidas deben ser, por un lado, pocas, y, por otro lado, lo más sencillas posibles, eso sí, dotadas de toda la semántica y funcionalidad necesarias.

Este requisito no es más que una extensión del ya visto en el capítulo anterior aunque aquí se refiera al conjunto de abstracciones necesarias para representar al objeto y su computación.

Ofrecer mecanismos de gestión sencillos

Por otro lado, el entorno de ejecución sobre el que se ejecuten las aplicaciones de usuario debe ser capaz, no sólo de ofrecer una abstracción de alto nivel de la máquina sino también debe ofrecer los mecanismos necesarios para gestionarla de forma sencilla y logrando el máximo partido de la máquina subyacente[Nut97].

Flexibilidad del entorno

Este modelo de computación busca, ante todo, introducir la concurrencia en el sistema con el objetivo de mejorar el rendimiento por encima de los límites de la máquina subyacente. Estos intentos se han llevado a cabo de distintas formas con distintos enfoques y han tenido diferente éxito.

Sin embargo, en general, diferentes aplicaciones tienen diferentes requisitos no funcionales. De esta forma, hacer el sistema flexible, significa dotar al sistema de una implantación flexible que pueda cambiar según la variación de los requisitos de cada aplicación [WS95].

Eficiencia

Al igual que en los entornos tradicionales, uno de los requisitos más evidentes es el de la eficiencia. Se trata pues de sacarle el máximo partido al hardware subyacente, aunque se busca también un compromiso entre eficiencia y conveniencia, es decir, el usuario puede estar dispuesto a perder algo de eficiencia a cambio de un entorno más conveniente de ejecución.

Interfaz adecuada

Por último, los mecanismos ofrecidos por el sistema operativo para la ejecución de sus métodos deben ofrecer una interfaz adecuada a los objetos de usuario, lo cual implica que, dado que el único modo de trabajo es la OO, todos los mecanismos ofrecidos por el sistema operativo (planificador, elementos para la sincronización como los semáforos, etc.) deben ser objetos que responden a mensajes enviados por otros objetos.

2 Integración de la computación en el paradigma de Orientación a Objetos: Un Reto Difícil

La integración de la concurrencia en los sistemas orientados a objetos, a pesar de ser una línea de investigación muy prometedora [Nie89, Pap89] presenta algunas dificultades derivadas de la interacción con algunas características intrínsecas de los modelos de objetos que no se dan en los sistemas puramente secuenciales.

Los mecanismos orientados a objetos, como las clases o la herencia, y los mecanismos de concurrencia, como los hilos o los cerrojos, proporcionan dos dimensiones de estructuración del software distintas, que el desarrollador de aplicaciones OO debe tomar en consideración simultáneamente. Trabajos como [MWY90, MY90, MY93], entre otros, han detectado la necesidad de un esfuerzo considerable por parte de los programadores para evitar conflictos a la hora de desarrollar software teniendo en cuenta ambas dimensiones.

Esto ha provocado la aparición de numerosos trabajos para diseñar modelos OO concurrentes, que integren características OO con la concurrencia y la sincronización necesarias, eliminando la necesidad de considerar dos dimensiones separadas al desarrollar software OO concurrente.

2.1 Dificultades: Problemas de Integridad y la Anomalía de la Herencia

La construcción de un entorno de computación orientado a objetos es una tarea difícil en la que surgen algunos problemas debidos, fundamentalmente, a la interferencia entre el modo operacional, en el que las relaciones entre objetos son las que se representan por el paso de mensajes entre ellos, y el modelo composicional, que soporta unas relaciones cliente/servidor muy distintas, tales como la herencia o la agregación.

2.1.1 El problema de la Integridad del Objeto

En un sistema secuencial, donde existe un único hilo de control, la característica de encapsulación es suficiente para proteger el estado interno de los objetos frente a manipulaciones arbitrarias, asegurando de esta forma la consistencia del mismo.

La introducción de la concurrencia en un entorno de objetos debe hacerse de tal forma que se evite la violación del estado interno de los objetos. Y, para ello, debe hacerse con la colaboración de los objetos. La abstracción de un objeto como la encapsulación de un conjunto de servicios y estado, debe tratarse de forma muy delicada, en presencia de la concurrencia.

En el caso más simple, en el que los objetos sean meros contenedores de datos, como en la mayoría de los lenguajes OO, la encapsulación se ve violada por los intentos de múltiples clientes concurrentes que no sincronizan sus accesos.

Recíprocamente, la encapsulación de los objetos que actúan como clientes también se ve comprometida si se le carga con la responsabilidad de sincronizar explícitamente sus peticiones a objetos compartidos.

Esto nos llevará a concluir, al final de este capítulo la conveniencia de algún tipo de modelo de objetos activo.

2.1.2 El Problema de la Herencia

Otro problema que surge de la integración de OO y concurrencia, es el famoso **problema de la herencia** o **Anomalía de la herencia** [MWY90, MY93].

En este caso, la dificultad estriba en que el comportamiento heredado puede ser muy sensible a la ejecución concurrente de métodos. En general, puede ser muy difícil extender el comportamiento heredado de forma consistente, de tal forma que se viola la encapsulación de la superclase por la necesidad de exponer detalles de implantación y la reutilización se ve comprometida dado que los métodos heredados no pueden sincronizarse con los métodos de la subclase.

2.2 Aproximaciones para la Integración de Concurrencia y OO

Las aproximaciones seguidas para integrar la concurrencia en entornos OO varían considerablemente con respecto a la relación entre la abstracción de **objeto** y aquella definida para representar la computación en el entorno, que denominaremos **actividad**[1], y también con respecto a lo que la abstracción de objeto definida es capaz de soportar. Un detallado análisis de las distintas posibilidades se encuentra en [Pap89]. En cada caso, los objetos pueden ser considerados entidades concurrentes, tipos de datos abstractos pasivos y compartidos o procesos.

Algunos entornos OO definen los objetos como entidades totalmente independientes de la computación, de tal modo que no están dotados de ningún mecanismo o propiedad predefinida para hacer posible la ejecución concurrente de sus métodos de forma controlada. El programador debe, con ayuda de los mecanismos ofrecidos por el

---

[1] Se ha elegido el nombre de actividad en detrimento de proceso o hilo, como nombre neutro, para no dar la sensación de sugerir una implementación concreta para tal entidad.

entorno, diseñar aplicaciones en las que los métodos de los objetos se ejecuten de forma adecuada.

En el otro extremo, es posible definir objetos que, por sí mismos, están dotados de características relacionadas con la computación concurrente de sus métodos. Por ejemplo, sistemas como Merlin[AK95], Apertos[Yok92] definen los objetos como entidades de computación puramente secuenciales que serializan la ejecución de sus métodos y, por tanto, no tienen concurrencia intra-objetos. Implementan un modelo de objetos activo donde el objeto ejecuta métodos de forma secuencial, de entre los solicitados.

Sin embargo, esto también impone una considerable merma en el nivel de concurrencia permitido en el sistema.

En función de cuál es la semántica del objeto respecto a la computación, podemos establecer distintas aproximaciones[Pap89, PK90, Nie87].

2.2.1 Aproximación Ortogonal

Esta primera aproximación, considera que los objetos son entidades pasivas, meras contenedoras de código, totalmente independientes y ajenas a la computación.

En este caso, dado que la computación es totalmente independiente de la noción de objeto, la sincronización también lo es y el sistema operativo debe ofrecer mecanismos como semáforos o cerraduras para sincronizar las invocaciones concurrentes a métodos de un mismo objeto.

Esta aproximación no ofrece ninguna protección implícita al estado de los objetos, ante la ejecución de métodos por parte de las actividades. Por tanto, o bien los compiladores modifican la semántica del objeto a discreción, utilizando el resto de los mecanismos ofrecidos por el sistema operativo para dotarlo de propiedades que lo protejan de los accesos concurrentes, o bien deben ser los programadores los que, utilizando los medios ofrecidos por el sistema operativo y el compilador, codifiquen los objetos clientes de forma que supongan la ejecución en un ambiente concurrente ya que, en caso contrario, su ejecución puede ser correcta en un ambiente puramente secuencial y fracasar ante invocaciones concurrentes.

La aproximación ortogonal tiene serios inconvenientes. Por un lado, presenta **problemas de integridad**, al no ser suficiente la protección ofrecida por la encapsulación en el caso de ejecuciones concurrentes,

El hecho de que la ejecución correcta de los métodos dependa, en determinados entornos de ejecución, del modo en que se implementan, es contrario al principio de abstracción, lo que **limita la reutilización de los objetos**.

De este inconveniente, se deriva la limitación en la autonomía de los objetos dado que los clientes deben sincronizarse unos con otros antes de acceder a un determinado objeto servidor.

Entre los lenguajes y entornos de programación OO que se ajustan a esta aproximación están Emerald[BHJ+87], Smalltalk-80[GR83] o Guide[DKM+89].

Las siguientes aproximaciones extienden la noción de objeto para que este tenga en cuenta la computación, fundamentalmente, la ejecución concurrente de sus métodos. Se basan pues en dotar al objeto de un mayor contenido semántico, integrando en el propio objeto ciertas características o propiedades relacionadas con la computación,

fundamentalmente con la ejecución de sus métodos como la gestión de las invocaciones concurrentes, de tal forma que la definición de objeto incluye los mecanismos necesarios para que estos puedan proteger su estado interno de la ejecución concurrente de sus métodos.

2.2.2 Aproximación Heterogénea

La aproximación heterogénea proporciona dos tipos de abstracciones, los objetos activos y los pasivos.

Mientras los **objetos activos** u **objetos concurrentes** están dotados de mecanismos que protegen su estado interno frente a las ejecuciones concurrentes de sus métodos, los **objetos pasivos** son meros contenedores de métodos, al igual que en el caso anterior, y su estado interno se protege automáticamente dado que sólo pueden ser utilizados dentro de objetos activos.

En esta aproximación podemos incluir ConcurrentSmalltalk[YT87], que soporta tanto objetos Smalltalk ordinarios como objetos atómicos que serializan la invocación concurrente de sus métodos, o Argus[LS82], que ofrece *guardians* y *clusters*.

Frente a la aproximación ortogonal ofrece varias ventajas. Por un lado, proporciona un mejor soporte a la abstracción de objeto, dado que los objetos pueden ser utilizados en cualquier entorno, sin necesidad de preocuparse de si su implantación soporta la concurrencia o no. Por otro, favorece la reutilización de clases preexistentes, dado que no es necesario determinar si la implantación de una clase soporta la invocación concurrente de métodos.

Su mayor inconveniente es la necesidad de dos clases distintas de objetos, los objetos activos y los pasivos, con una semántica muy distinta. Esto limita la reutilización del código, ya que es difícil intercambiar dos implantaciones distintas de un mismo objeto.

2.2.3 Aproximación Homogénea

Esta última aproximación, extiende también la noción de objeto para tener en cuenta su entorno de computación, supuestamente concurrente pero, a diferencia de la aproximación heterogénea, todos los objetos son objetos activos y tienen control sobre la sincronización de sus peticiones concurrentes.

Esta aproximación da lugar a objetos más pesados, inconveniente que, si bien puede resultar importante en el caso de objetos pequeños, recargados con la sobrecarga que supone la maquinaria de ejecución, también es cierto que existen varios mecanismos para lograr que esta sobrecarga no sea mayor que la de la anterior aproximación.

La ventaja fundamental de esta aproximación es la uniformidad en la semántica del objeto, lo que favorece que el programador no tenga que preocuparse de decidir si un objeto debería ser implementado como un tipo de datos o como un objeto concurrente.

Esta es la aproximación más interesante por cuanto la ventaja fundamental que ofrece es un pilar básico del modelo de objetos del Sistema Operativo propuesto en esta tesis, por lo que, a continuación, se refinará aún más el modelo homogéneo.

En el caso de que los objetos tengan ciertas propiedades relacionadas con la concurrencia, estas son, como mínimo, la protección de su estado interno respecto a la ejecución concurrente de sus métodos. Sin embargo, estas propiedades pueden

extenderse para incluir también los mecanismos de computación en sí, dando lugar a dos aproximaciones.

Aproximación No Integrada

La ejecución de un método se expresa explícitamente, por medio de objetos que representan los hilos de control, que se comunican y sincronizan a través de la invocación de métodos en los objetos. En este caso, los objetos controlarían el acceso a los métodos, sincronizando las ejecuciones de los distintos hilos.

Aproximación Integrada

Los objetos se consideran entidades activas, con toda la semántica que conlleva, de tal forma que la ejecución se expresa por la creación e interacción de los objetos y la sincronización forma parte del objeto.

Esta aproximación tiene la ventaja de integrar en una única abstracción, tanto la representación como la computación, dando lugar a una entidad autónoma, a la vez que simplifica el flujo de control por la interacción de las distintas entidades existentes. La comunicación y sincronización de los objetos está claramente definida.

Es por tanto, la aproximación más prometedora de todas y la que se adoptará como modelo para el entorno de computación del Sistema Operativo propuesto en esta tesis. En el capítulo X, se realizará un estudio más detallado del mismo.

El ejemplo más representativo de esta aproximación es ABCL/1[YST+87] en la que los objetos son entidades activas que se comunican por medio de paso de mensajes y que encapsulan una actividad interna.

Capítulo VIII

Requisitos Funcionales para el Sistema de Computación de un SOOO

El sistema de computación del sistema integral orientado a objetos debe cumplir una serie de requisitos funcionales comunes a la mayoría de los sistemas operativos tradicionales y de investigación.

Deberá aislar al usuario de la máquina subyacente ofreciéndole la abstracción o abstracciones necesarias que representen el poder computacional de la máquina. De esta forma, el usuario puede acceder a los recursos de la misma de forma sencilla y logrando el máximo rendimiento posible.

1 Proporcionar Abstracciones adecuadas

En los capítulos III y VII se analizó la integración de la computación en el entorno OO que se propone en esta tesis desde dos puntos de vista.

Mientras en el capítulo III se discutió la inconveniencia de mantener las abstracciones tradicionales conviviendo con la nueva abstracción de objeto, en el capítulo VII, se parte de la base de proporcionar una abstracción única, propuesta indiscutible de los capítulos V y VI. A partir de esto, se presentan varias aproximaciones para unificar las propiedades de representación y de computación del objeto, aunque, al final, sólo una de ellas parece conveniente.

1.1 Semántica del Objeto respecto a la Computación

Es posible definir dos modelos de objeto respecto a la computación, que se diferencian, fundamentalmente, en la potencia de la abstracción de objeto que ofrecen, dotándola, en uno de los casos, de un contenido mucho mayor.

Aunque, podría encajar perfectamente aquí la discusión entre ambos modelos, se deja esta para el capítulo X, con el fin de no alargar demasiado este capítulo. Baste aquí con presentar someramente, los principios básicos del modelo propuesto en esta tesis.

El modelo computacional propuesto, define al objeto como una **entidad autocontenida** tanto en el sentido de representación de la entidad del mundo real, en cuyo caso encapsula datos y métodos, como en el sentido de la computación, en cuyo caso, encapsula la capacidad de ejecución de sus métodos y gestión de las actividades internas, a la vez que se le dota de un mecanismo de comunicación unificado [YT88a].

Los objetos son, por tanto, **agentes de procesamiento autónomos**, con vida independiente y capacidad de comunicación[WY88, YSH+90, YTT89].

El objeto se convierte en la **unidad de procesamiento**. Cada uno de ellos, además de tener una zona de memoria local persistente en la que almacenar su estado, está dotado de su propio conjunto de procesadores virtuales, lo que le confiere un poder de computación individual independientemente de otros objetos, y tienen un ciclo de vida conceptualmente muy simple. basado en recibir mensajes y procesarlos [Nie93, JBV93].

Este poder de computación individual puede ser **serie** (un único procesador virtual por objeto), en cuyo caso, el objeto ejecuta los métodos solicitados en alguna secuencia, o **concurrente** (varios procesadores virtuales), en cuyo caso, el objeto debe tomar decisiones acerca de cuándo y cómo llevar a cabo los métodos solicitados. Así, por ejemplo, Apertos[Yok92], recibe los mensajes y los secuencializa.

El problema se centra entonces en cómo diseñar estos agentes autónomos de procesamiento, qué función lleva a cabo cada agente de computación, cómo se llevan a cabo varias actividades en paralelo y cómo deben interactuar los agentes. La noción de objeto surge como el camino más prometedor.

1.2 Modelo de Objetos para la concurrencia

La representación en tiempo de ejecución de cada objeto debe incluir la representación de la ejecución de los distintos métodos que se hayan solicitado, así como aquellos que, por alguna razón, todavía no han podido ser ejecutados.

Dado que el sistema integral propuesto está compuesto por una máquina abstracta y un sistema operativo, el modelo de computación está íntimamente ligado a los mecanismos de computación que ofrezca la máquina, que se verán aumentados y complementados por aquellos ofrecidos por el sistema operativo.

Por tanto, será necesario definir la representación y los mecanismos de computación tanto a nivel del sistema operativo como de la máquina abstracta para construir el objeto definido en el apartado anterior, intentando siempre que el modelo nos permita lograr el máximo nivel de concurrencia tanto entre objetos como dentro de un objeto y que además garantice la consistencia del estado interno del objeto.

1.2.1 Máquina abstracta

Una posibilidad sería que la máquina abstracta proporcionase una abstracción de objeto de dos tipos, el tipo objeto dato y el tipo objeto actividad y que el sistema operativo se encargase de ofrecer los mecanismos de gestión de ambas abstracciones.

Sin embargo, en esta tesis se propone una solución distinta en la que es la propia máquina abstracta la encargada de ofrecer la abstracción de objeto como agente de procesamiento.

Esta decisión se ha tomado por dos razones fundamentales. En primer lugar, para dar soporte en el nivel más básico a los principios fundamentales propugnados en esta tesis. En segundo lugar, por una cuestión de eficiencia.

Así pues, es la máquina abstracta OO la que se encarga de ofrecer la representación en tiempo de ejecución de los objetos, así como de proporcionar la herramienta básica para la computación en el entorno integral que no es otra que el paso de mensajes.

La representación de cada objeto se compone de sus datos y sus métodos así como un contexto, similar a otras representaciones como Merlin[AK95] que represente cada

mensaje todavía no procesado y un contexto para cada método en ejecución que representa el estado del mismo.

1.2.2 Sistema Operativo

Sería sencillo implantar en la máquina abstracta un modelo de objetos para la computación fijo e inamovible. Es decir, definir qué características se desean en el objeto con relación a la ejecución de sus métodos e implantar en la máquina abstracta una abstracción de objeto de ese tipo. Por ejemplo, podría implantarse un objeto tipo Muse[YTM+91] que serialice la ejecución de sus métodos.

Sin embargo, la codificación del modelo a bajo nivel en la máquina, sin posibilidad de modificación, hace que el sistema integral sea inflexible.

Por tanto, se opta por permitir que el Sistema Operativo modifique el comportamiento de la máquina abstracta, permitiéndole modificar ciertas características del modelo de objetos definido por la máquina.

Modelo de Objetos en el Sistema Operativo

El modo en que el sistema operativo modifica el modelo de objetos de la máquina no puede ser otro que representando cada característica como un objeto o conjunto de objetos, si se desea mantener la abstracción única y homogénea.

Este conjunto de objetos, formarán el **entorno de ejecución del objeto** definiendo el modo en que este se comportará y ejecutará sus métodos. Se asocia con el objeto formando una entidad en sí, aunque no indisoluble, y dotándolo de un mayor contenido semántico y de los mecanismos necesarios para ofrecer un comportamiento determinado.

Así, cada objeto modela, mediante objetos del Sistema Operativo el **mecanismo de comunicación**, especificando una política de aceptación ante los mensajes enviados por otros objetos.

Un objeto también puede, ayudado de los objetos del Sistema Operativo, modelar su **concurrencia interna**. De esta forma, determina su comportamiento ante la recepción y aceptación de mensajes y la gestión de los mismos en función de las características del objeto y de su estado actual. Puede servir los mensajes inmediatamente, rechazarlo o retrasar su ejecución de acuerdo a su estado y a cualesquiera circunstancias que puedan influir en la ejecución.

Cada objeto también puede especificar, mediante un conjunto de objetos del sistema operativo, la política de planificación concreta sobre aquellos de sus métodos cuya ejecución no haya finalizado todavía.

Por último, es imprescindible que cada objeto sea capaz de enviar respuestas al objeto que le había solicitado el servicio. Aunque no es necesario que estas respuestas sean los resultados de la ejecución correcta y completa del método solicitado, sino que puede suceder que se transmita la excepción resultante de algún problema en la ejecución.

Por tanto, la arquitectura del sistema operativo difiere sensiblemente de las arquitecturas tradicionales donde el sistema operativo estaba formado por módulos claramente identificados y separados de las aplicaciones.

En este caso, las funciones tradicionales del sistema operativo son completamente ofrecidas por los objetos que colaboran con la máquina modificando el modelo de objetos que define y algunos de los mecanismos que ofrece.

1.2.3 Máquina Abstracta + Sistema Operativo = Soporte a objetos

De esta forma, la computación esta integrada dentro de los propios objetos, es más, es un componente fundamental de los mismos de forma que el procesamiento queda completamente encapsulado dentro del objeto y es totalmente transparente al exterior.

Al integrar todos los conceptos en una única abstracción este modelo difiere notablemente de los demás modelos en los que se ofrecían distintas abstracciones (procesos, tareas, procesos ligeros, hilos de usuario, *scheduler activations*).

Un elemento fundamental de esta tesis es la forma en que Máquina y Sistema Operativo cooperan para definir el entorno de ejecución de los objetos. Aunque esto se desvelará más adelante, fundamentalmente en los Capítulos XI, XII y XIII, podemos adelantar aquí que la técnica propuesta se presentó ya en el Capítulo II. Se trata de la **Reflectividad**.

2 Interacción entre objetos: Mecanismo de comunicación uniforme

El mundo de objetos ofrecido por el entorno integral estará constituido por un conjunto de objetos que, evidentemente, necesitarán interaccionar entre sí para llevar a cabo tareas complejas.

2.1 Modo de trabajo exclusivamente Orientado a objetos

En un sistema integral donde la única abstracción existente es el objeto y el único modo de trabajo aceptable es OO, tal y como se señaló en los Capítulos V y VI, es bastante inmediato identificar el mecanismo de interacción entre objetos con la invocación a métodos entre objetos que se llevará a cabo mediante el **paso de mensajes**.

Comunicación y Flujo de Control

De esta forma, tanto la comunicación como el flujo de control en este sistema de computación OO, en el que los propios objetos son las entidades de procesamiento, se llevan a cabo a través de un mecanismo de comunicación integrado con el modelo de objetos.

Se representa por una serie de invocaciones a métodos (paso de mensaje) entre objetos que no sólo transmiten información entre objetos, sino que activan el procesamiento de métodos del objeto invocado transmitiendo así el flujo de control[YT88a].

Introducción de la Concurrencia en el sistema

En un entorno en el que el único modo de interacción es el envío de mensajes, este se convierte también en el modo de introducir concurrencia en el modelo. Si todo paso de mensajes involucrase una transferencia de control y un posterior retorno del mismo

al objeto origen de la llamada, la ejecución de métodos en el entorno sería puramente secuencial.

Ante la recepción de un mensaje, un objeto ejecuta el método asociado y produce el resultado solicitado. Mientras tanto, el objeto origen del mensaje puede, **suspender su ejecución** a la espera del resultado o **continuar su ejecución en paralelo** con el método solicitado. Esto introduce la concurrencia entre objetos[Nie93] que se verá más en detalle en el apartado 3.

Por tanto, es necesario determinar el mecanismo de paso de mensajes de forma que no sólo sirva para comunicar objetos, sino que se convierta en el medio de introducir o aumentar el nivel de concurrencia en el sistema e incluso pueda suponer un mecanismo de sincronización entre objetos.

2.2 Uniformidad en la Comunicación

El mecanismo de comunicación debe ser uniforme, es decir, la invocación de un método en un objeto debe ser transparente al hecho de que el objeto sea local o remoto al usuario, utilizando el mismo mecanismo en cualquiera de los dos casos[Nie89].

AGRA, el sistema de distribución del sistema integral se encargará de lograr este punto[ATD+98a, ATD+98b].

2.3 Modelos de Paso de Mensajes

Existen distintos modelos de invocación de métodos, cada uno de los cuales introduce un mayor grado de concurrencia en el modelo y, por ende, un mayor rendimiento adoleciendo, sin embargo, de una complejidad creciente[Car89, Car93].

2.3.1 Modelo Síncrono

Es el mecanismo de invocación de métodos más sencillo posible. En este modelo, cuando un objeto origen invoca un método en un objeto destino, la actividad del objeto origen se bloquea hasta que la operación se ejecute completamente en el objeto destino y se retorne el resultado de la operación al objeto origen.

Este modelo es el utilizado en los sistemas orientados a objetos secuenciales e incluso algunos sistemas OO concurrentes tienen como único modelo de comunicación la invocación síncrona (Emerald[BHJ+86], Java[GM96]).

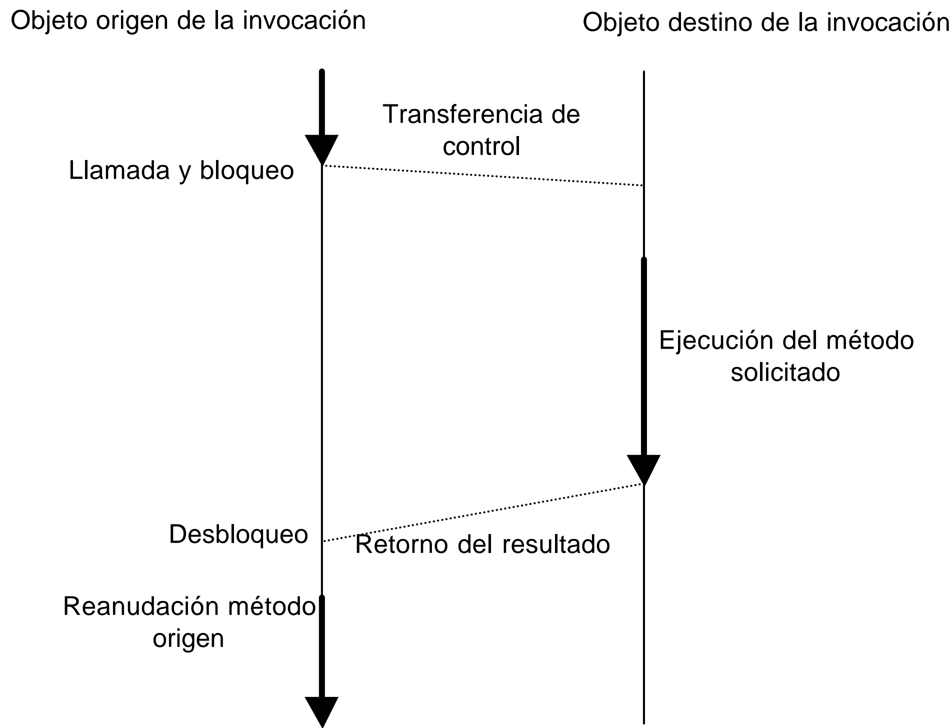


Figura 8.1 Flujo de Control en una Invocación a Método Síncrona

2.3.1.1 Ventajas

Sencillez

Este modelo de invocación de métodos es, con mucho, el más sencillo posible. Supone la introducción de una única instrucción de paso de mensajes que activa la ejecución del método solicitado en el objeto llamado y hace innecesaria la instrucción receive ya que el control se cede de forma explícita al objeto destino.

Ampliamente extendido en los modelos de los lenguajes

Una de las ventajas adicionales del paso de mensajes síncrono es la familiaridad de programadores con este modelo de comunicación ya que, en la gran mayoría de los lenguajes de programación se lleva a cabo las acciones de forma estrictamente secuencial.

Buen rendimiento en ambientes secuenciales

El paso de mensajes síncrono, fuerza la ejecución secuencial debido a la cesión de control que se realiza en cada invocación. Esto hace que, en el caso de ambientes puramente secuenciales, se adapte perfectamente, sin introducir ningún elemento adicional innecesario.

Baja sobrecarga de implantación

En cuanto a la implantación, se trata de suspender y reanudar la ejecución de un método, por lo que la sobrecarga introducida por la gestión de las invocaciones síncronas es muy baja y, en ambientes distribuidos, será despreciable frente al tiempo de envío real del mensaje.

En caso de ambientes secuenciales, dará lugar a una implantación muy eficiente ya que encaja completamente el modelo de alto nivel.

2.3.1.2 Inconvenientes

Aunque son muchas las ventajas que ofrece, presenta un inconveniente de muy difícil solución.

Limitación del nivel de concurrencia

Debido a la cesión de control realizada al invocar el método en el objeto destino, la ejecución en el objeto origen se suspende hasta la terminación de dicho método, haciendo difícil el paralelismo entre varias actividades.

Aunque en algunos casos sea esto lo que realmente se desee, existen muchos otros casos en los que la ejecución origen podría continuar en paralelo durante mucho o todo el tiempo restante.

Por tanto, este mecanismo no puede ser el único en nuestro sistema ya que, dado que la concurrencia se introduce por la comunicación entre objetos, eliminaría la concurrencia definiendo un entorno de computación puramente secuencial.

2.3.2 Modelo Asíncrono

El modelo de comunicación o invocación de métodos asíncrono es otro de los modelos utilizados tradicionalmente para comunicar dos entidades.

En este modelo, la invocación de un método por parte del objeto origen no supone la transferencia de control absoluta al objeto destino sino que, una vez enviado el mensaje, el control se devuelve al objeto origen.

Este continua su ejecución en paralelo con la ejecución del método invocado en el objeto destino hasta que necesite sincronizar su ejecución con el objeto destino, bien porque necesite el resultado de la invocación o por alguna otra razón.

Eso supone que el objeto origen debe invocar explícitamente alguna primitiva de sincronización. Esta primitiva provocaría la suspensión de su ejecución hasta que el objeto destino termine la ejecución del método solicitado y devuelva el resultado (en caso de que haya que devolver resultado).

Sin embargo, en el envío o invocación asíncrona también es posible que el origen no espere ningún resultado del destino, en cuyo caso ambas ejecuciones continúan en paralelo indefinidamente y la sobrecarga que introduce es incluso menor que el caso de envío síncrono ya que no hay ni que parar ni que reanudar al objeto origen.

Algunos sistemas OO soportan este modelo de comunicación para incrementar la concurrencia, ya que se evita que el emisor del mensaje tenga que esperar el resultado completo de la ejecución.

Será necesario introducir algún mecanismo de planificación para determinar los turnos de ejecución de las distintas actividades, los criterios que se siguen para determinar los turnos, y, en general, los detalles típicos de los sistemas en los que puede haber más de una tarea a realizar simultáneamente.

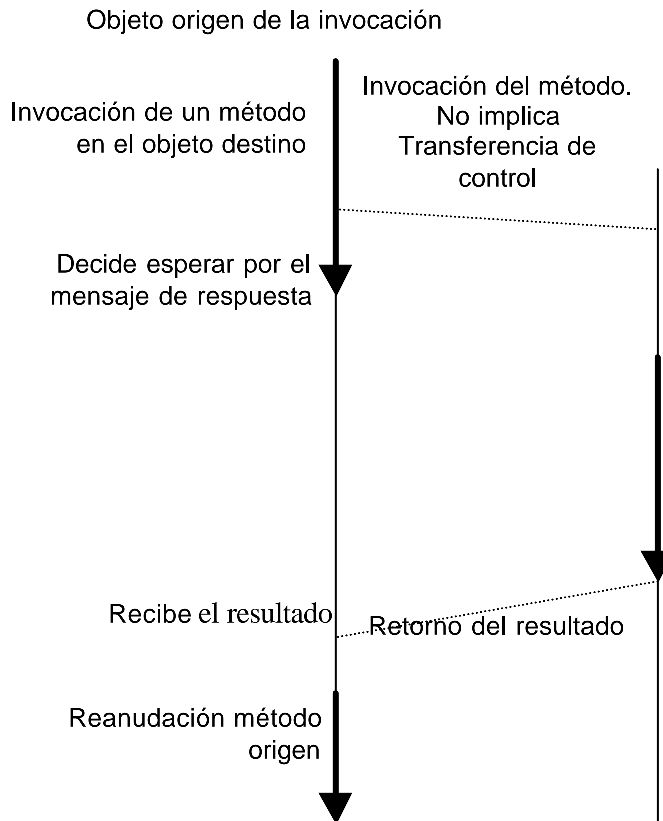


Figura 8.2 Flujo de Control en una Invocación a Método Asíncrona

2.3.2.1 Ventajas

Introduce Concurrencia

Al continuar la ejecución del objeto origen en paralelo con la del objeto que sirve el mensaje, se aumenta el grado de concurrencia, favoreciendo un mejor rendimiento[SG98].

2.3.2.2 Inconvenientes

Nueva Primitiva de Comunicación

Es evidente que, a diferencia del caso síncrono, existen ciertas dificultades para utilizar este modelo de invocación de métodos como mecanismo de sincronización. Por tanto, en el caso de que el objeto origen desee sincronizar su ejecución con el objeto destino, es necesario introducir una nueva primitiva, **wait**, para sincronizar la ejecución del objeto origen y el objeto destino.

Complica el modelo de Programación

La ausencia de sincronización *per se* del modelo, supone que, a la hora de sincronizar dos objetos, es necesario añadir código explícito de sincronización, utilizando para ello la nueva instrucción añadida.

Esto supone añadir cierta complejidad a la programación a la hora de sincronizar dos objetos, debido al aumento de código al tener que añadir, explícitamente, la instrucción de sincronización.

2.3.3 Modelo Wait-By-Necessity

Este modelo de comunicación combina el nivel de concurrencia del modelo asíncrono con la simplicidad conceptual del modelo síncrono. La semántica del modelo es semejante a la invocación asíncrona de forma que, la invocación de un método en un objeto por parte de otro, no supone la transferencia de control del segundo al primero, sino que ambas ejecuciones continúan en paralelo simultáneamente[RWB97].

En la versión más típica de este modelo de comunicación, después de enviar el mensaje, la ejecución del método en el objeto origen continúa en paralelo con la ejecución del método invocado.

El objeto origen recibe como resultado de la operación una referencia a un objeto o, en algunos casos, un objeto de un tipo especial, Cbox en ConcurrentSmalltalk[YT87a], Objetos Future de Java//[Car93], y otros, que contendrá el resultado de la invocación.

El modelo wait-by necessity modifica ligeramente la semántica de la invocación síncrona tradicional, ya que la sincronización se establece cuando el objeto origen intenta utilizar el resultado, invocando un método en la referencia del objeto resultado, en cuyo caso, si todavía no está instanciado al resultado, el objeto origen suspende su ejecución hasta que este hecho se produzca.

La respuesta final tiene el efecto de desbloquear a todos aquellos objetos que estaban intentando enviarle un mensaje a una referencia no instanciada todavía y hace que ese objeto sea ahora el invocado. Cualquier mensaje posterior a dicho objeto no sufrirá ningún bloqueo.

De esta forma, la respuesta se hace al objeto respuesta y no al invocador original. De hecho, no se conoce la identidad de tal objeto.

La referencia del objeto resultante puede pasarse como argumento en invocaciones a otros objetos que podrían bloquearse también. Es decir, este mecanismo puede emplearse para sincronizar a más de dos objetos.

Transparencia de los objetos Futuros

Los objetos futuros son totalmente transparentes al usuario.

- Se mantiene una compatibilidad absoluta con la ejecución secuencial del código lo que significa que, por supuesto, un método nunca debe "ver" los objetos futuros.
- No requieren ninguna modificación del código que invoca o es invocado, sino que se crean automáticamente. Esto es posible porque los objetos futuros suelen ser una subclase del objeto retorno.

Es uno de los modelos de programación más prometedores como lo demuestra el creciente interés de distintos proyectos en llevar a cabo el paso de mensajes siguiendo este modelo [AK95, CV98] lo que nos ha llevado a incorporarlo a nuestra máquina, como se verá en el Capítulo XV.

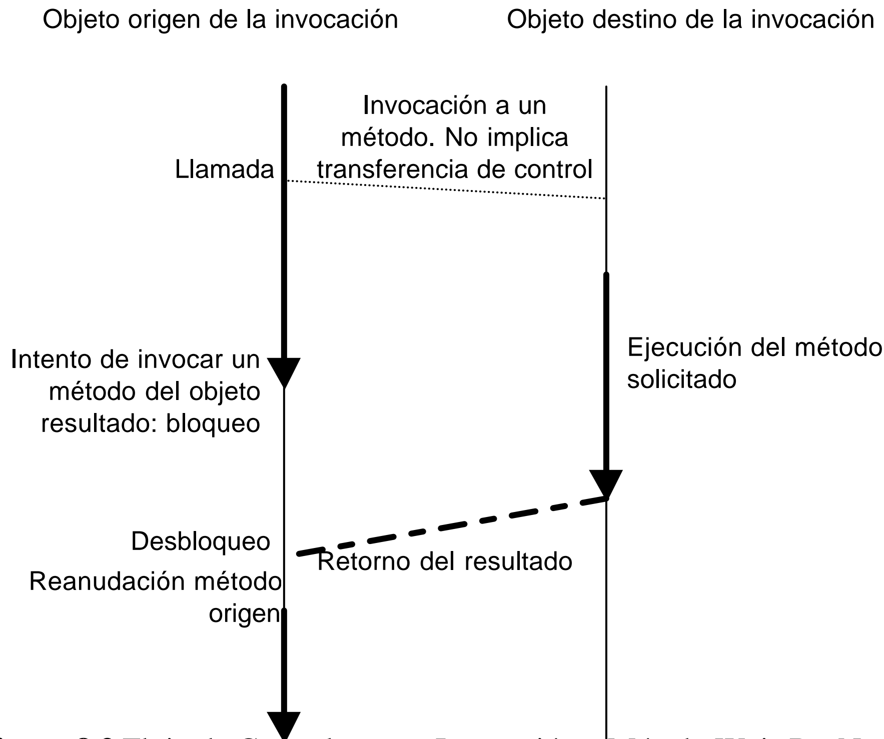


Figura 8.3 Flujo de Control en una Invocación a Método Wait-By-Necessity

2.3.3.1 Ventajas

Aumento del nivel de concurrencia

La ventaja más destacable de este modelo de invocación de métodos es el aumento del nivel de concurrencia frente al modelo de invocación síncrono.

Sin embargo, y en contra del modelo asíncrono, no implica el conocimiento por parte del programador de mecanismos diferentes del modelo síncrono ya que toda la lógica del envío-respuesta se hace internamente y, por tanto, la concurrencia queda completamente oculta al programador.

Sencillez

Se le oculta la complejidad adicional de la concurrencia al usuario ya que el modelo de programación puede ser el mismo y la existencia de objetos no instanciados todavía lo resolverá, en nuestro caso, la máquina abstracta de forma totalmente transparente.

Introduce poca sobrecarga adicional respecto al modelo síncrono

No introduce prácticamente sobrecarga en la gestión ya que el desbloqueo de los objetos que se queden bloqueados en el envío de un mensaje a un objeto resultado de una invocación, supone poco incremento del coste frente a la gestión de los bloqueos en las invocaciones síncronas.

Adecuada para ambientes paralelos y distribuidos

Adecuada para su extensión a máquinas multiprocesadoras y ambientes distribuidos.

Eficiencia

Trabajos como [CV98] muestran cómo la invocación de métodos wait-by-necessity supone una mejora del rendimiento, fundamentalmente en ambientes paralelos y distribuidos.

Es más, la computación a través de Internet se ve muy favorecida debido a la superposición automática de las computaciones y comunicaciones.

2.3.3.2 Inconvenientes

Complicación del modelo de excepciones

El inconveniente más importante que presenta este modelo es la gestión de las excepciones dado que un método puede producir una excepción que se gestionará cuando métodos posteriores ya se hayan servido o, al menos, solicitado.

Por ejemplo, si intentamos calcular algo como v=(y/z)+factorial(x), en un modelo síncrono, con un único hilo de ejecución activo en cada momento, si z llega a valer 0, la aplicación completa se percatará del error por una propagación directa de las excepciones.

Sin embargo, en el modelo wait-by-necessity, en el que ambas acciones se llevan a cabo en paralelo, es posible que la aplicación solicite la evaluación de la segunda parte de la expresión, mientras se esté llevando a cabo la primera que, está claro que va a resultar errónea debido a que z=0.

Debido a esto, el modelo de excepciones se complica para gestionar este tipo de problemas.

3 Gestión de la Computación Concurrente

La definición de objeto como entidad **autónoma** y **autocontenida** – que encapsula su computación–, y la adopción del **paso de mensajes** como mecanismo de transferencia de control y de introducción de la concurrencia, introducen la visión del entorno como uno en que distintos objetos llevan a cabo distintas actividades de forma independiente, **concurrencia entre objetos**, a la vez que un objeto puede estar llevando a cabo distintas actividades, **concurrencia intra-objeto** [Nie93, CC91, BG96, GKS94].

Transparencia

La transparencia de actividad o de ejecución, oculta el hecho de que métodos invocados en un mismo objeto se ejecutan simultáneamente en hilos separados así como el hecho de que el método origen y destino se ejecutan en paralelo. Ambos hechos deben ser totalmente transparentes a los programadores.

Dentro de la gestión de un entorno concurrente, son dos los requisitos fundamentales que el modelo de computación debería lograr de manera transparente a cualquier otro componente del sistema[Nie93], gestión de la computación interna de forma transparente a los clientes y gestión de la computación externa.

3.1 Concurrencia interna

Un objeto se define como una entidad autónoma, con capacidad de procesamiento independiente de otros objetos. Esta capacidad de procesamiento se activa debido a la recepción de un mensaje solicitando la ejecución o servicio de un método.

Cada objeto puede recibir un número indefinido de mensajes en un corto período de tiempo de forma que deba procesar mensajes sin haber terminado de procesar los anteriores, simultáneamente.

Esto introduce la **concurrencia intra-objetos** que aumenta la eficiencia del sistema, y facilita su incorporación en el futuro a máquinas multiprocesadoras y ambientes distribuidos.

3.1.1 Control de la Concurrencia Interna: Sincronización

La presencia de actividades concurrentes en un mismo objeto requiere cierto grado de sincronización, es decir, cierta maquinaria adicional al objeto que garantice una correcta ejecución de los métodos asegurando la propiedad de encapsulación.

Es necesario que el modelo contemple la necesidad de manejar las distintas actividades que se ejecutan dentro de un mismo objeto gracias a mecanismos de control de la aceptación y la computación de solicitudes dentro del objeto. El propio mecanismo de paso de mensajes puede ser de utilidad para lograrlo.

Existen varias aproximaciones dependiendo de la forma en que los objetos gestionan su concurrencia interna[Nie93]. Tales aproximaciones pueden incorporarse *de facto* al entorno haciendo que todos los objetos creados en el mismo tengan el mismo comportamiento, o puede crearse un entorno en el que puedan definirse objetos de varios tipos.

Objetos secuenciales

Estos objetos tienen su propio y único hilo de ejecución y, por tanto, gestionan los mensajes de forma serie.

Sistemas como Apertos[Yok92], su antecesor Muse[YTM+91] o Merlin[AK95], definen un objeto que secuencializa sus métodos lo que no sólo proporciona una abstracción más sencilla, sino también un mecanismo efectivo y robusto para el control de la concurrencia a cambio de limitar el grado de concurrencia[MHM+95].

Objetos quasi-concurrentes

Los objetos gestionan su concurrencia interna multiplexando explícitamente múltiples hilos de control. Pueden procesar varios mensajes concurrentemente pero, como mucho puede haber uno ejecutándose en un momento específico de tiempo. Un ejemplo puede ser Hybrid[Nie87].

Objetos concurrentes

Su propia definición (y por tanto su implantación) soportan hilos concurrentes simultáneamente. Tales hilos pueden crearse implícitamente ante la recepción de un mensaje o explícitamente.

3.1.2 Entorno propuesto en esta tesis

En el entorno propuesto en esta tesis, se opta por la última opción. El sistema integral, máquina más sistema operativo, permitirá al usuario definir el modo en que un objeto servirá los mensajes: secuencial o concurrente, en cuyo caso deberá aportarse información acerca de las restricciones a la hora de servir los mensajes.

Transparencia

Esta sincronización o control interno, debe ser totalmente transparente a los clientes, que no necesitan conocer ni las restricciones ni a los demás clientes, de modo que debe ser posible reemplazar una implantación secuencial de un objeto activo con una implantación concurrente del mismo objeto sin afectar a los clientes.

Por tanto, cada objeto debe tener la posibilidad de ejecutar sus métodos de forma concurrente sin necesidad de que el cliente deba especificar qué condiciones deben darse para que el objeto lleve a cabo ese método.

Además, el objeto debe gestionar su concurrencia de forma que no sea necesaria ninguna sincronización explícita por parte de los clientes[Nie93, VBJ+96].

3.2 Concurrencia Externa

Dado que el sistema se proporciona como un mundo de objetos independientes que interaccionan, debe garantizarse la posibilidad de que varios objetos puedan estar ejecutando métodos simultáneamente.

Debido a que la interacción entre objetos tiene lugar cuando un objeto envía un mensaje a otro (invocación de métodos) y vistos los distintos modelos, se refuerza la tesis de que el modelo síncrono no puede ser el modelo base de nuestro entorno, ya que merma considerablemente el nivel de concurrencia introducido.

Debido a ello, y como ya se vio en el apartado 2.3.3, la concurrencia externa se basará en la introducción del modelo wait-by-necessity.

3.3 Sincronización

La presencia de actividades concurrentes en un mismo y distintos objetos, requiere cierto grado de sincronización, restricciones para asegurar la correcta ejecución de los métodos. La sincronización puede asociarse a los objetos y sus medios de comunicación[BGL98].

3.3.1 Sincronización en el paso de mensajes

Cualquiera de los dos modelos de comunicación descritos suponen la sincronización, en algún momento, de la ejecución de emisor y receptor.

En el caso síncrono, esta se produce de forma inmediata dado que, para reanudar su ejecución, el emisor del mensaje debe esperar a que el destino termine la ejecución del método invocada y retorne el resultado.

En el caso asíncrono, la sincronización se produce por necesidad, cuando el objeto origen pretende hacer uso del resultado de la invocación. También se llama algunas veces sincronización perezosa.

En cualquiera de los dos casos, el paso de mensajes puede servir perfectamente para la **sincronización entre objetos**.

La identificación de sincronización con paso de mensaje, tiene la ventaja de asegurar una parte importante de la sincronización, de forma transparente.

3.3.2 Sincronización a nivel de objeto

Otro aspecto de la sincronización es el que se refiere a la gestión de las distintas actividades en ejecución dentro de un mismo objeto.

Para ello, el objeto debe expresar qué circunstancias son adecuadas para la ejecución de un determinado método.

Una posibilidad es que la máquina abstracta ofrezca un modelo de objetos serie, en el que los métodos se ejecutan uno a uno, de acuerdo a un orden. Sin embargo, puede ser necesario un control más global de la concurrencia.

Mecanismos de más alto nivel

Para lograrlo, se solicita la intervención del sistema operativo, que modifica el comportamiento por defecto de la máquina permitiendo al objeto codificar las condiciones necesarias para la ejecución de un método. Cualquier intento de ejecución de un método, será filtrado por el Sistema Operativo.

Reutilización del código secuencial

Al separar el código de sincronización del código secuencial, se permite la reutilización de ambos, pudiendo adaptarlos a otras condiciones de ejecución.

4 Planificación

La planificación de actividades es una tarea tradicional de cualquier entorno de computación que, usualmente, se lleva a cabo a través de una entidad denominada **Planificador** que ejecuta una **Política de Planificación** utilizando mecanismos de planificación de más bajo nivel como el cambio de procesos o cambio de contexto[SG98, Sta98, TW98].

En este caso, la aproximación propuesta es básicamente la misma. La planificación será llevada a cabo por un objeto Planificador (del Sistema Operativo) que utilizará los mecanismos ofrecidos por la máquina abstracta (intercambio de objetos).

Sin embargo, existe una pequeña diferencia con los entornos tradicionales, al tener dos niveles de actividades concurrentes, las actividades internas a un objeto y el conjunto global de actividades.

En sistemas operativos tradicionales multihilo que implementaban dos modelos de planificación (a nivel usuario y a nivel kernel) se encontró una situación similar.

En este caso, la librería de hilos a nivel usuario se encarga de planificar los hilos de usuario sobre la entidad kernel para su ejecución lo que proporcionaba cierta flexibilidad al ser una librería en modo usuario.

4.1 Planificación interna

Cada objeto debe tener la posibilidad de planificar sus actividades internas organizando su ejecución en base a criterios que pueden ser particulares de cada objeto concreto.

Sin embargo, es necesario lograr este requerimiento de forma totalmente transparente al mundo de objetos exterior. Para una gestión compatible con la idea de abstracción única es imprescindible lograr:

Planificación de peticiones

Cada objeto debe estar dotado de un mecanismo que le permita planificar el servicio de peticiones basándose en el conocimiento de su propio estado interno. Deberá por tanto, ser capaz de parar (*local delays*) y reanudar ejecuciones en función de las circunstancias del objeto o el sistema. La gestión de estos retrasos locales debe ser totalmente transparente al cliente.

Planificación de respuestas

Cuando durante la ejecución de alguno de sus métodos, un objeto haga una petición a otro objeto, es posible que esta petición implique espera o no quedando en manos de cada objeto el determinar la forma en que se piden servicios a otro objeto. En cualquier caso, es necesario que el objeto sepa reaccionar ante la petición de un servicio y ante la llegada de una respuesta ofreciendo la posibilidad de parar y reanudar (*remote delays*) ejecuciones en función del conocimiento de su estado interno. De la misma forma que en el caso anterior, la gestión de los retrasos remotos debe ser completamente transparente al proveedor del servicio.

4.2 Planificación externa

La máquina subyacente es un recurso que debe ser compartido por todas las actividades que se estén llevando a cabo en cada momento de tiempo. Tradicionalmente, los sistemas operativos dedicaban una parte de su código a gestionar el reparto de tiempo y a dicha parte la denominaban planificador o *scheduler.*

Los sistemas operativos tradicionales controlan el reparto de los recursos (entre ellos el tiempo de CPU o de ejecución) utilizando un esquema de planificación fijo o, en algunos casos, este esquema puede ser variable de forma estática (en el momento del arranque) o dinámica.

Normalmente, aquellos procesos que tienen igual prioridad se planifican en modo round-robin y cada uno de ellos disfruta de un quantum de tiempo. Si aparece algún proceso más prioritario puede arrebatar la CPU al proceso actual aunque este último aún no haya terminado de usar su tiempo.

De la misma forma, cuando varios objetos compiten por la máquina subyacente, debe establecerse algún mecanismo para determinar los turnos de ejecución. En el caso de multiprocesadores, este mecanismo será mucho más complejo porque habrá que tener en cuenta factores como la sincronización entre objetos, los recursos que un objeto tiene que puede necesitar el otro, prioridades, etc.

5 Modelo de excepciones

Las excepciones son un concepto que está orientado fundamentalmente a la realización de programas robustos mediante el manejo estructurado de los errores.

5.1 Lanzar y Gestionar Excepciones

Es necesario que el sistema ofrezca un mecanismo de gestión de excepciones. Este mecanismo proporciona los protocolos que permiten al programador establecer la comunicación entre la rutina que detecta la situación excepcional y las entidades que la gestionan, es decir, permiten a los usuarios elevar excepciones y definir manejadores.

Elevar Excepciones

Elevar una excepción supone:

- **Identificar** la situación excepcional
- **Interrumpir** la secuencia usual de ejecución

Buscar el gestor para la excepción

Invocar al gestor pasándole la información relevante

Gestionar Excepciones

Los gestores se definen en (o asocian con) una entidad para la gestión de una o varias excepciones y se invocan cuando la excepción ocurre.

Gestionar la excepción supone llevar a cabo las acciones necesarias para devolver al sistema a un estado coherente, es decir,

- **Reanudación**: Transferir el control a la sentencia adecuada después de la que causó la excepción, o
- **Terminación**: Desechar el contexto entre la sentencia que causó la excepción y la sentencia a la que está asociado el gestor, o bien
- **Propagación**: Elevar una nueva excepción

5.2 Gestión de excepciones con el paradigma de la orientación a objetos

5.2.1 Representación de las excepciones: Clases

Las excepciones son entidades complejas cuya descripción puede verse considerablemente beneficiada en un entorno orientado a objetos. [Don90].

Clase Excepción

La forma más directa para representar las excepciones en un entorno OO, es crear una clase excepción de la que derivan las excepciones concretas.

Cada subclase define los métodos de gestión de la excepción correspondiente de forma que, cuando se produzca una excepción, se ejecuta el método de gestión de la excepción de la clase correspondiente a la excepción producida.

Ventajas

De esta forma se utilizan las ventajas derivadas de la orientación a objetos:

- Se pueden crear nuevas excepciones como subclases de la clase Excepción y no existe ninguna distinción entre excepciones definidas por el usuario y excepciones definidas por el sistema de forma que todas ellas se elevan y manejan de la misma forma.
- Definir las excepciones como clases posibilita el organizarlas en una jerarquía que hace que el sistema de excepciones sea fácilmente extensible y reusable.

5.2.2 Gestión de las excepciones

Las excepciones serán objetos instancias de alguna clases Excepción, que informarán de determinadas condiciones en la ejecución de una o varias instrucciones en un método de un objeto.

Cada objeto puede instalar o no manejadores para una excepción. De esta forma, la gestión de una excepción puede hacerse de forma local en el objeto que provocó la excepción o puede propagarse hacia atrás a los objetos que invocaron al que provocó la excepción (de forma similar al modelo de procesos).

La propagación de excepciones debe diseñarse teniendo en cuenta la posibilidad de un entorno distribuido de forma que sea completamente transparente.

Esto se ve muy favorecido por el mecanismo de comunicación basado en la invocación de métodos en objetos y el retorno de objetos resultado que ofrece la posibilidad de unificar el mecanismo de retorno de resultados con el mecanismo de propagación de excepciones como se verá más adelante en el capítulo XV.

Capítulo IX

Requisitos No Funcionales para el Sistema de Computación de un SOOO

Los requisitos no funcionales determinan los parámetros a los que hay que ajustarse a la hora de proporcionar la funcionalidad vista en Capítulo VIII. El grado en que el modelo propuesto se ajuste a los requisitos no funcionales determinará la bondad del sistema.

Entre ellos se encuentran requisitos generales a la hora de diseñar y construir cualquier sistema operativo, como pueden ser el rendimiento, y otros específicos para el diseño y construcción de un sistema integral Orientado a objetos como la homogeneidad o el modelo único de objetos.

En este capítulo nos centraremos únicamente en aquellos requisitos no funcionales que se refieren a la computación. Otros ya se comentaron en el Capítulo VI.

1 Modelo único de Objetos también para la Computación

De la introducción de la concurrencia en el modelo básico de objetos y del requerimiento de modelo único ya mencionado en el Capítulo VI, se deduce la necesidad de que todos los objetos tengan un comportamiento esencialmente idéntico respecto a la concurrencia, aunque pueden tener especializar su comportamiento respecto a la misma [ATA+98].

Así, un objeto se define como una entidad activa, con capacidad de ejecutar sus métodos de forma autónoma e independiente de los demás. Por tanto, todos los objetos tienen la capacidad de gestionar múltiples ejecuciones de varios de sus métodos simultáneamente pero es posible restringir un objeto para que las secuencialice.

Un ejemplo claro de modelo único es Apertos[Yok92].

Sin embargo, otros sistemas como Guide[CFH+93] define **objetos** (*instance-objects*) y **tareas** (*tasks*). La única forma de poder acceder y manipular un objeto es a través de una tarea, de tal forma que un objeto debe formar parte del espacio de direcciones de una tarea para poder invocar sus métodos.

Al no ser posible compartir objetos entre tareas, se asegura la protección del objeto por aislamiento.

2 Abstracción única y homogénea

Aunque ya en el Capítulo VI se hizo referencia a la propiedad de abstracción única y homogénea que debía cumplir el sistema operativo propuesto en esta tesis, se vuelve otra vez sobre ello, en referencia esta vez a la vertiente computacional de la abstracción única, es decir, a la necesidad de unificar la semántica de los objetos para proporcionar una única abstracción con poder de ejecución.

2.1 Integración de Conceptos

El Sistema Integral Orientado a Objetos propuesto, promueve al objeto como **abstracción única** [MHM+95, BGL98] también desde el punto de vista de la computación, aglutinando lo que, en etapas anteriores del desarrollo de Sistemas Orientados a Objeto, se dispersaba, bien en varias abstracciones con semántica muy distinta como procesos o hilos y objetos, bien en una abstracción, el objeto, pero existiendo varios tipos de objetos con comportamiento diferente.

En el primer caso, los objetos eran meros contenedores de datos y métodos, y el sistema operativo debía proporcionar mecanismos de protección de su estado interno frente a injerencias externas al objeto. Clouds[DLA+91] o Guide[BLR94] son ejemplos de esta tendencia.

En el segundo caso, se proporcionan varios tipos de objeto, los objetos de datos y los objetos actividades. Mientras los primeros se comunican mediante el paso de mensajes o invocación a métodos, los segundos lo hacen a través de la ejecución de métodos en objetos de datos compartidos. Choices[CJM+89] es un ejemplo claro.

La integración de conceptos convierte al objeto en la única abstracción ofrecida por el sistema integral.

Esta abstracción, heredable por las capas superiores del mismo, con una semántica común aunque pueda ofrecer un comportamiento particular, deriva en un modelo de programación en el que no existen conceptos separados para datos y procesos, sino que los objetos en sí mismos son entidades dotadas de capacidad de procesamiento.

2.2 Encapsulación de la Computación

Los objetos se convierten en la unidad de procesamiento, capaces de ejecutar sus métodos, ante la solicitud por parte de otro objeto. La computación o procesamiento en este sistema integral OO se representa como una secuencia de mensajes entre objetos. Dado que un objeto es una entidad auto-contenida y dotada de un protocolo de comunicación unificado, la descomposición de un sistema en un conjunto de objetos resulta muy flexible.

Es más, en un sistema OO donde todo está encapsulado en forma de objetos (entidades auto-contenidas) y la única forma de acceder a un objeto es solicitando un servicio que él mismo debe proporcionar y gestionar (invocación a métodos), la propia semántica del objeto protege a los mismos de accesos ilegítimos, dando soporte a una interacción ordenada entre objetos [YTT88a].

Con esto, gran parte del significado del proceso o tarea pierde su sentido ya que esta entidad servía como elemento de protección del espacio de direcciones frente a otros

procesos que podían, incluso intencionadamente, intentar realizar un acceso ilegítimo. Los autores de TUNES [~Tun] las califican de abstracciones paranoicas.

La definición de entidad auto-contenida no sólo es suficiente para asegurar la no intromisión externa sino que, a la vez, garantiza la propiedad de encapsulación .

2.3 Abstracción homogénea

Al igual que en el capítulo VI se discutió la necesidad de homogeneidad de objetos en aspectos como la granularidad o la diferencia entre objetos del lenguaje y objetos del sistema, también en lo que se refiere a la computación, la homogeneidad debe se una propiedad a cumplir.

Así, todos los objetos del sistema tienen la misma semántica, todos ellos se definen como entidades de procesamiento autónomas y no es posible establecer una división en el conjunto de los objetos utilizando la capacidad de computación como discriminante[Nie89].

Ni siquiera los objetos del sistema son capaces de ejecutar más cosas o de forma diferente del modo de ejecución de los objetos normales de usuario. Todos los objetos son capaces de ejecutar sus métodos ante la invocación de los mismos por parte de otros. Aunque, evidentemente, cada objeto puede controlar la ejecución interna de sus métodos atendiendo a diferentes criterios.

Esta simplicidad conceptual hace que todo el sistema en su conjunto sea fácil de entender, y se elimine la desadaptación de impedancias al trabajar con un único paradigma. La tradicional distinción entre los diferentes elementos de un sistema: hardware, sistema operativo y aplicaciones de usuario se difumina.

3 Visión Uniforme del sistema

La integración de la computación en el sistema integral orientado a objetos mediante la abstracción de objetos autónomos auto-contenidos, además de lograr reducir al máximo el número de abstracciones necesarias, propicia la representación de las funciones y propiedades de las entidades físicas o conceptuales en el dominio del problema como simples objetos. De esta forma, los objetos son las entidades básicas de gestión y las aplicaciones se definen como un conjunto de objetos y la interacción entre ellos.

De este modo, la descomposición del sistema en una colección de objetos es muy flexible y la estructura resultante del sistema queda muy natural, proporcionando a los programadores y usuarios una visión única del sistema como un conjunto de objetos [YMF+91, YTT89].

La utilización de las funciones y propiedades de los objetos se lleva a cabo gracias al envío de mensajes al objeto. Cuando un objeto recibe un mensaje, si el mensaje es aceptable, comienza una secuencia de acciones con el fin de dar servicio a la invocación.

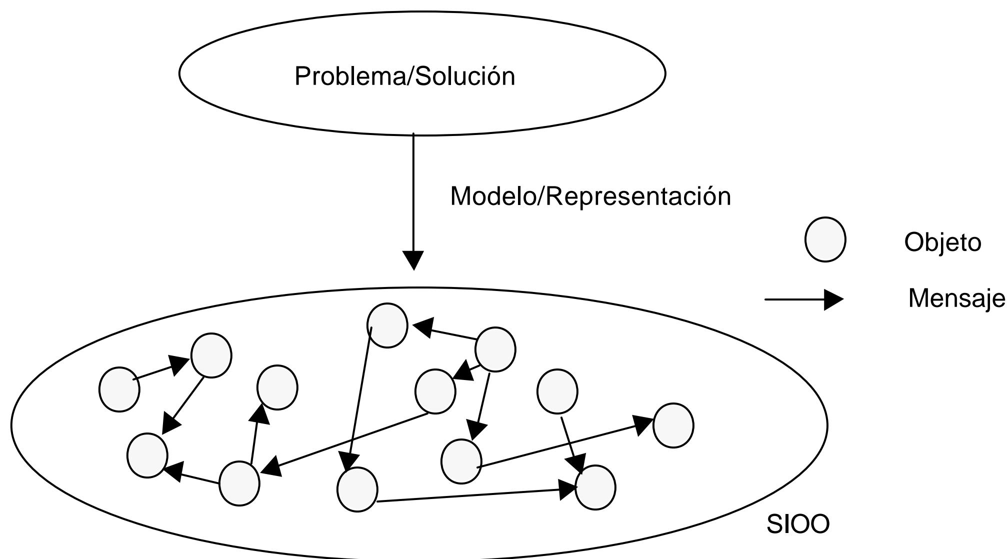


Figura 9.1 Representación del problema real en objetos que se comunican

Esta idea de crear objetos activos o concurrentes como la unidad software de estructuración del sistema más pequeña, previene problemas de sincronización y encaja de forma natural con el entorno OO[IYT95].

Esta es una aproximación bastante natural que tiene algunos antepasados como los actors[YSH+90].

3.1 Entorno sencillo y Economía de Conceptos

La uniformidad proporcionada por la definición de objeto como entidad no sólo estructural sino de procesamiento, supone el diseño de un entorno de computación mucho más sencillo, en el que la economía de conceptos se lleva al límite, al ofrecer una única abstracción.

3.2 Abstracción potente

El hecho de ofrecer una abstracción única y homogénea proporciona multitud de ventajas (entorno sencillo y uniforme, economía de conceptos, ..) y, por el contrario, una mayor complejidad inherente a la propia abstracción.

Esta necesidad de una abstracción mucho más potente y compleja, hace que el esfuerzo invertido a la hora de diseñarla sea mayor, con el fin de que la integración de su computación no dé al traste con la sencillez mencionada.

El sistema operativo debe ofrecer, no sólo soporte a la existencia de objetos, sino también mecanismos de gestión de la computación integrados en los mismos, de forma que las capas superiores tengan una visión muy simple del sistema como un mundo de objetos con vida independiente que interaccionan entre sí enviándose mensajes.

Así, cada objeto ofrecerá una funcionalidad al exterior que únicamente podrá solicitarse a través del envío de un mensaje de petición por parte de cualquier otro objeto del mundo de objetos que actuará de cliente.

El mecanismo de computación que da soporte a la abstracción de objeto, se encarga de ofrecer mecanismos que permitan la ejecución de forma controlada de los métodos de los objetos.

4 Mecanismo de computación flexible

4.1 Sistemas Tradicionales: Poco adecuados para la OO

El capítulo III concluyó con el rechazo de los entornos tradicionales como capa básica para la construcción de un Sistema OO. Este rechazo se produjo, básicamente por dos razones.

Abstracciones poco adecuadas

En los sistemas operativos tradicionales, el sistema de computación ofrecía a las aplicaciones abstracciones tipo proceso o hilo que representaban la computación de la máquina, a la vez que proporcionaban interfaces para la planificación, la sincronización y la concurrencia.

Sin embargo, en muchos casos, las aplicaciones requieren niveles de funcionalidad y rendimiento que los sistemas operativos son incapaces de proporcionar. Este problema se hace aún más patente cuando se intenta construir un sistema OO sobre un sistema tradicional.

Así, por ejemplo, los procesos y los hilos son muchas veces inadecuados y, en algunos casos insuficientes: los procesos son entidades demasiado pesadas, existe un problema de falta de integración entre los hilos de usuario y los hilos del sistema, el comportamiento de tales abstracciones no encaja con la OO, etc.

Falta de flexibilidad en los mecanismos

Otro problema de los Sistemas Operativos tradicionales es la falta de flexibilidad a la hora de permitir que las aplicaciones personalicen el sistema para adecuarse a sus requisitos.

En cuanto a la computación, la planificación es un claro ejemplo. La planificación de los sistemas operativos formaban parte integral del mismo y era bastante rígida.

Los sistemas operativos tradicionales controlan el reparto de los recursos (entre ellos el tiempo de CPU o de ejecución) utilizando un esquema de planificación fijo. Normalmente, aquellos procesos que tienen igual prioridad se planifican en modo round-robin y cada uno de ellos disfruta de un quantum de tiempo. Si aparece algún proceso más prioritario puede arrebatar la CPU al proceso actual aunque este último aún no haya terminado de usar su tiempo.

Frecuentemente, por no decir siempre, han soportado únicamente una única política de planificación y además, esa política era fija e inamovible. Algunas veces se proporcionan variantes de esta política básica, como varias clases de planificación a las que se asignan hilos con diferentes propósitos. Sin embargo, incluso estas variantes suelen estar integradas en la implantación del sistema y no pueden ser adaptadas fácilmente a las necesidades de las aplicaciones individuales.

Esto provoca un grave problema de integración con las aplicaciones actuales que, en la mayoría de los casos son aplicaciones multihilo y, en muchos casos necesitan ser capaces de definir una estrategia particular para sus hilos (servidores de ficheros, servidores de red, ...)

Los sistemas operativos micronúcleo significaron una apertura hacia la posibilidad de no integrar todas y cada una de las funciones de un sistema operativo en el núcleo, lo que significaba la imposibilidad de actuar sobre ese mecanismo.

El sistema operativo Mach[Bla90] implementa una política de planificación de tiempo compartido. Cuando un procesador comienza a ejecutar un hilo, a ese hilo se le asigna un quantum de tiempo. Cuando este expira, se replanifica. También puede suceder que el hilo termine o se bloquee antes.

El rendimiento de aquellas aplicaciones que ejecutan más de un hilo, se puede ver beneficiado de aquella información que el usuario proporciones al planificador a la hora de tomar decisiones.

El planificador de Mach se implementa en el kernel del sistema pero permite a los usuarios proporcionar pistas que influencien sus decisiones[Bla90]. Estas pistas son de dos clases:

- **Desacuerdo**: Indica que el hilo actual no debería ejecutarse. Puede ser Suave si se trata de dar el procesador a otro hilo si es posible, fuerte, si se le da el procesador a otro hilo y decrementa la prioridad temporalmente o absoluta, en cuyo caso se bloquea.
- **Handoff**: Indica que un hilo específico debería ejecutarse en lugar del actual

Existen sistemas operativos orientados a objetos como Choices[RJC88] que diseñan una jerarquía de planificadores y se puede instalar cualquiera pero sólo uno.

4.2 Máquina abstracta OO + Sistema Operativo OO: Separación de política y mecanismos

Al intentar diseñar sistemas operativos extensibles es importante separar las decisiones políticas de los mecanismos que las implementan[Rus91b].

De la misma forma que para lograr la portabilidad es necesario aislar las dependencias de la arquitectura subyacente en módulos que luego puedan ser cambiados, las distintas políticas con un mismo ámbito de aplicación, pueden ser identificadas y modularizadas.

El aislamiento de la política se consigue gracias a los mecanismos de herencia y polimorfismo que soportaban la portabilidad. Estos mecanismos permitían la definición de interfaces con múltiples implantaciones.

Las decisiones de política con interfaces características pueden representarse con clases abstractas que definen su interfaz. Clases concretas pueden implementar la interfaz para políticas específicas.

Esto permite que existan diferentes políticas según las necesidades del momento cambiándose gracias al reemplazo de un módulo concreto por otro. El polimorfismo permitirá que sea la instancia de la política concreta que se haya instanciado, el que ejecute el mensaje.

Un ejemplo claro de la aplicación de esta técnica es la planificación en sistemas operativos.

El mecanismo es muy simple. Procesadores ociosos necesitan un medio para conseguir algo que hacer. La decisión política involucrada es la ordenación de las tareas pendientes siguiendo un algoritmo concreto.

Otro ejemplo es la gestión de memoria virtual. La decisión de qué página echar de memoria puede variar aunque la interfaz del paginador permanezca inalterable.

Por otra parte, el diseño de un SIOO formado por máquina abstracta OO y Sistema Opeativo OO promueve todavía más la separación de política y mecanismo. Esta separación permitirá, como se verá la modificación dinámica de la política.

4.2.1 Máquina Abstracta: Los mecanismos de bajo nivel

La máquina abstracta, además de dar soporte a la abstracción de objeto, también ofrece los mecanismos de bajo nivel para su comunicación, planificación de la ejecución de sus métodos, etc.

Sin embargo, no dice nada acerca de cómo usar esos mecanismos.

4.2.2 Sistema Operativo: Las políticas de Gestión

Es el sistema operativo el que, a más alto nivel implementa las distintas políticas de comunicación, planificación, sincronización, etc. de los objetos.

Para ello, el sistema operativo ofrece, en forma de objetos, los distintos elementos que un objeto necesita para gestionar su computación: planificador, comunicador, distintos objetos que definan el modelo de objetos para la concurrencia del objeto, etc.

Estos definen la política a seguir por el objeto y la implementan utilizando los mecanismos básicos ofrecidos por la máquina.

El hecho de que los objetos del sistema operativo sean objetos normales, como los objetos de usuario, permite a las aplicaciones cambiar un objeto por otro, incluso en tiempo de ejecución[1]. Esto hace que sea posible la modificación dinámica del entorno de los objetos, sobre una plataforma común.

5 Eficiencia

Aunque este es un primer diseño del sistema de computación, es necesario que se haga de forma que se logre la mayor eficiencia posible.

Las decisiones que, fundamentalmente, han influido a la hora de definir un sistema eficiente son las siguientes.

- Basar el mecanismo en una máquina abstracta que lleve a cabo, en último instancia las acciones que se llevan a cabo mayor número de veces, como el paso de mensajes. Al estar los mecanismos básicos de comunicación y ejecución de métodos codificados en la máquina abstracta, estos se vuelven mucho más eficientes.

[1] Esto será posible, siempre y cuando la aplicación tenga permisos para ello y el nuevo objeto responda a la misma interfaz.

- Ofrecer un paso de mensajes asíncrono, que se ve muy favorecido por el boom de la computación a través de internet, permitiendo, si no eliminar, si disminuir los periodos de latencia.
- Economía de Conceptos. Ofrecer una única abstracción que englobe al recurso, el fichero ejecutable y el proceso, supone un considerable ahorro de tiempo a la hora de diseñar una aplicación compleja.

Capítulo X

Modelo de Objetos para el Sistema de Computación

El diseño de un sistema de computación Orientado a Objetos implica, fundamentalmente, la definición de las características encarnadas en cada objeto, es decir, la definición de un modelo de objetos, fundamentalmente en aquellas propiedades que más de cerca involucren el aspecto de objeto como agente de computación.

Aunque es relativamente sencillo definir un modelo que cumpla los requisitos funcionales definidos en el capítulo VIII, lo realmente complejo es definir un modelo que cumpla además los requisitos no funcionales expresados en el capítulo IX.

1 Modelo de Objetos Activo versus Modelo de Objetos Pasivo

Existen gran cantidad de trabajos [Nie87, CC91, Pap89] que estudian los dos modelos de objetos más relevantes en lo que se refiere a la computación en el sistema.

A más alto nivel, la diferencia fundamental entre ambos, estriba en la **uniformidad** o no del modelo, es decir, en la definición de una semántica común a todos los objetos o, por el contrario, en la existencia de clases de objetos distintas con diferentes propiedades y comportamiento.

En el caso de estar en la primera posibilidad, estaremos ante un **modelo de objetos activo**, que ofrece objetos homogéneos y el soporte al objeto encapsulará la ejecución de sus métodos.

La segunda posibilidad recibe el nombre de **modelo de objetos pasivo**, y se basa en ofrecer una clase de objetos especiales, representativa de la computación que serían el equivalente de los procesos en los sistemas operativos tradicionales.

Tradicionalmente, para incorporar las construcciones de concurrencia en un sistema convencional OO se introducía la noción de hilo de ejecución [DLA+91]. En este modelo, los objetos son entidades pasivas, meros contenedores de datos, y cada proceso o hilo puede ejecutar código del método de un objeto cualquiera en un momento dado.

Presentan ventajas como un buen rendimiento. Sin embargo, dado que los hilos se ejecutan independientemente, se presentan problemas de exclusión mutua cuando dos o más hilos intentan ejecutar simultáneamente ese método[MHM+95].

1.1 Homogeneidad: Un Objetivo Irrenunciable

En el capítulo IX se determinó como objetivo irrenunciable la homogeneidad de los objetos en todas las capas del sistema integral.

Como extensión natural de este objetivo, el diseño del mecanismo de computación para el sistema integral debe ajustarse también a él.

Esto conlleva la obligación de ofrecer el objeto como única abstracción en el nivel más básico del sistema integral OO y el mecanismo de computación debe diseñarse de tal forma que no viole esta regla.

Cumplir con este objetivo complica considerablemente el diseño del mecanismo de computación que deberá proporcionarse en forma de objetos como cualquier otro mecanismo del sistema, siendo sin embargo una abstracción mucho más primitiva.

2 Modelo de Objetos Pasivo

El modelo de objetos pasivo promueve la división de los objetos, desde el punto de vista de la computación, en dos grandes grupos que tienen características muy diferentes y que dan lugar a un mecanismo de computación muy similar al tradicional.

Un sistema que dé soporte a un modelo de objetos pasivo ofrece a las capas superiores dos tipos de abstracciones. Por un lado, los **procesos** y por otro, los **objetos**, y los programadores deben utilizar ambas a la manera tradicional para construir sus aplicaciones.

Puede suceder que el sistema subyacente abstraiga ambas entidades como objetos[RJC88]. Sin embargo, ambas abstracciones representan entidades con capacidades completamente distintas del sistema subyacente y su semántica difiere sustancialmente[CC91, Nie93, KB93].

2.1 Objetos Pasivos

El sistema de soporte los define como meros contenedores de datos y métodos de manipulación. Únicamente encapsulan un recurso que representa una entidad del mundo real y determinan los posibles accesos al mismo para acceder o manipularlo.

La comunicación con el resto de los objetos se realiza cuando un objeto proceso ejecuta una instrucción de paso de mensaje, que no es más que una forma de maquillar una llamada a procedimiento.

Su capacidad de ejecución es nula y dependen completamente de otros objetos para llevar a cabo los métodos.

2.2 Objetos Proceso

Los objetos proceso encapsulan la capacidad de computación de la máquina a nivel del sistema operativo y representan la computación que se lleva a cabo dentro del sistema ejecutando métodos ofrecidos por los objetos del tipo anterior.

Los objetos con esta definición representan un flujo de ejecución que puede englobar métodos ofrecidos por distintos objetos y, por tanto, tienen una vida independiente de los objetos por los que va pasando en su ejecución.

2.3 Semántica del Modelo

En este modelo, ambas entidades son completamente independientes y el sistema de soporte las representa en tiempo de ejecución de forma diferente y las dota de capacidades muy distintas.

Un proceso no está ligado ni restringido a un único objeto. En lugar de esto, un único proceso puede utilizarse para llevar a cabo todas las operaciones requeridas para satisfacer una acción. En consecuencia, un proceso puede ejecutar acciones dentro de varios objetos durante su vida.

Cuando un objeto proceso invoca un método en otro objeto, distinto al actual, su ejecución en el objeto en el que reside actualmente, se suspende temporalmente. Conceptualmente, el objeto proceso se mapea entonces en el espacio de direcciones del segundo objeto, donde ejecuta la operación apropiada.

Cuando completa esa operación, retorna al primer objeto, donde reanuda la ejecución de la operación original.

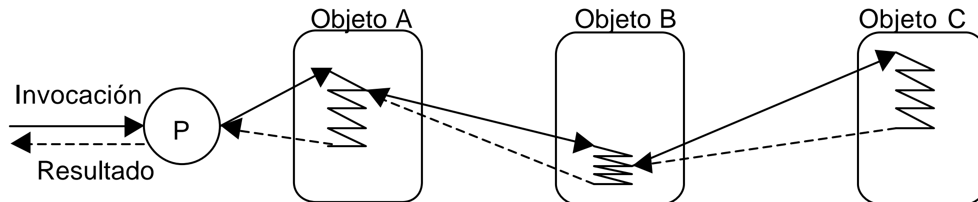


Figura 10.1 Ejecución de un método en el modelo de objetos pasivo

2.3.1 Interacción de Objetos

Un sistema OO que soporta un Modelo de Objetos Pasivo, típicamente, da soporte a un esquema de Invocación de Métodos Directa para soportar la interacción de objetos. En este modelo, un proceso es responsable de ejecutar todas las operaciones asociadas con una acción. Como resultado, un proceso migrará de operación en operación y, por tanto, de objeto en objeto, siempre y cuando la correspondiente acción ejecute una invocación.

Procesos y objetos residen en la misma máquina

Si un proceso invoca un objeto que reside en la misma máquina, las acciones serían las siguientes(Ver Figura 10.2):

1. El estado del proceso y el objeto en el que reside actualmente, se registran en la pila del proceso. El sistema puede proteger la pila para asegurar que esta información no pueda ser examinada o corrompida por actividades de invocaciones posteriores.
2. Los parámetros de la invocación se añaden a la pila.
3. El objeto invocado se carga en memoria y se realiza una **llamada a procedimiento** para comenzar la ejecución del código apropiado.
4. Cuando la operación termina, el resultado se retorna al cliente y el proceso se restaura al estado en el que se encontraba antes de la invocación.

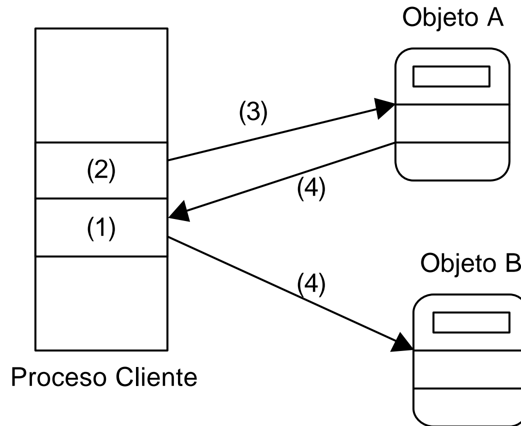


Figura 10.2 Interacción entre objetos y procesos en la misma máquina.

Procesos y objetos residen en máquinas distintas

Sin embargo, si el proceso invoca un objeto que reside en una máquina diferente, las acciones a ejecutar serían un poco diferentes. A continuación se muestran las tres nuevas acciones que habría que llevar a cabo. Las dos primeras, tras el paso 1, y la última, después del paso 3 (Ver Figura 10.3).

1.1. Se crea un mensaje que contiene los parámetros de la invocación y se envía a la máquina en la que reside el objeto destinatario.

1.2. La máquina destino que recibe el mensaje, crea un proceso que se ejecuta en nombre del proceso original.

3.1. Cuando la operación termina, se crea un mensaje que contiene el resultado de la invocación y se envía a la máquina en la que reside el proceso original. El proceso creado en la máquina destino, termina.

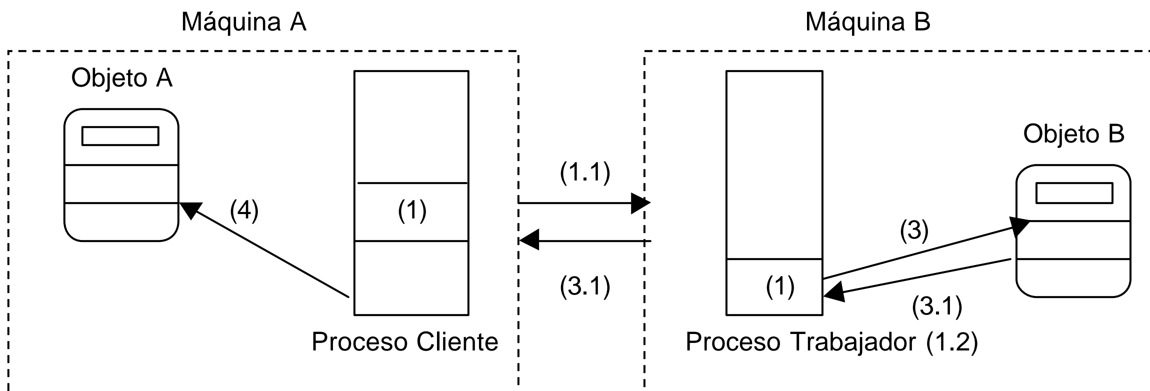


Figura 10.3 Interacción entre objetos y procesos en distintas máquinas.

Así pues, una invocación en un objeto local es similar a una llamada a procedimiento, mientras una invocación en un objeto remoto, es similar a una llamada a procedimiento remoto.

Este esquema incurre un una penalización del rendimiento menor que el caso del paso de mensajes cuando se trata de invocaciones locales. Sin embargo, cuando se trata

de invocaciones a objetos que no pertenecen a la misma máquina, este esquema tiene la sobrecarga adicional de crear y destruir procesos.

El lenguaje Java implementa un modelo de objetos pasivo en el que threads y objetos son entidades separadas. Como resultado, los objetos Java sirven como suplentes para la coordinación de los threads pero no abstraen ninguna unidad de concurrencia.

Esta relación entre objetos e hilos ha sido identificada en [AAS+98, VA98] como un factor que limita la utilidad de Java para la construcción de sistemas concurrentes.

Entre los sistemas de soporte a objetos más conocidos, Emerald[BHJ+86, BHJ+87] y Clouds[DLA+91] soportan un modelo de objeto pasivo.

3 Modelo de objetos activo

En un entorno secuencial, la secuencia de invocaciones de las operaciones está bajo el control de programador. Por su parte, en un entorno concurrente, se puede operar sobre un objeto a partir de diferentes objetos activos que pueden invocar cualesquiera operaciones en momentos arbitrarios.

En este contexto, la mejor aproximación es considerar al objeto como una entidad autónoma que encapsula su computación y que acepta o retrasa la ejecución de las operaciones de acuerdo a su estado interno, comunicándose con los demás mediante el envío de mensajes.

Se basa fundamentalmente en que el sistema de soporte sólo ofrece una entidad, el objeto, con una semántica y comportamiento únicos que, evidentemente, tendrá un contenido semántico mayor que en el modelo de objetos pasivo.

Un modelo de objetos activo es una extensión del paradigma cliente/servidor a un entorno orientado a objetos. Los objetos son entes autónomos que interaccionan con los demás objetos del entorno que le son conocidos.

3.1.1 Objetos Activos

La idea básica subyacente bajo el concepto de objeto activo es considerar a los objetos como entidades dotadas de sus propias actividades internas. El Sistema Integral Orientado a Objetos sólo ofrece una abstracción a las capas superiores, el objeto y la abstracción de objeto que soporta, tiene asociadas distintas actividades internas para gestionar invocaciones a métodos. Esta aproximación es simple y natural y tiene influencias en trabajos como [YT87].

El concepto de objeto activo es también un fundamento natural para construir agentes autónomos de más alto nivel, para sistemas distribuidos basados en el conocimiento.

Como ejemplo de sistemas de objeto que ofrezcan un modelo de objetos activo, podemos citar, entre otros, Amoeba[TRS+90], Argus[LCJ+87] o Chorus[RM87].

Al no existir entidades procesos separadas, los métodos de cada objeto no son accedidos directamente por un proceso origen. En lugar de esto, un objeto origen invoca un método en un objeto destino que acepta la invocación y ejecuta el método correspondiente valiéndose para ello de las actividades internas que el sistema de soporte le proporciona.

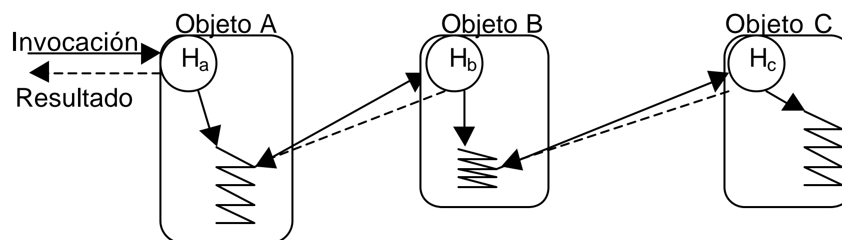


Figura 10.4 Ejecución de métodos en el modelo de objetos activo

La independencia de las actividades del objeto proporciona lo que, usualmente, se ha dado en llamar **Concurrencia Entre-Objetos**.

La ejecución de los métodos en los objetos activos puede llevarse a cabo de forma secuencial, lo que no implica que, cuando la ejecución de un método se para por alguna cuestión, no pueda ejecutarse algún otro método. Se habla en este caso de **objetos serie**. Apertos[Yok92] es un ejemplo de este caso.

En otros modelos como MetaJava[GK97b], se permite que un objeto activo procese varias invocaciones simultáneamente, con distintas actividades internas. Es lo que se denomina **Concurrencia Intra-objetos**. Esto incrementa el poder expresivo y la concurrencia global, aunque requiere un control mayor de la concurrencia para asegurar la consistencia del estado interno del objeto.

3.1.2 Interacción entre Objetos

Un Sistema Orientado a Objetos que soporte un modelo de objetos activo, habitualmente, soporta el paso de mensajes puro, como mecanismo de interacción entre objetos que ofrece transparencia en un entorno centralizado o distribuido respecto a la localización del objeto.

Cuando un cliente invoca un método en un objeto, se empaquetan los parámetros de la invocación en un mensaje, que se envía al objeto destino, basándose en su identificador global.

El objeto destino acepta el mensaje, desempaqueta los parámetros y ejecuta la operación especificada. Normalmente, cuando termina la operación, el resultado se empaqueta en un mensaje de respuesta que se envía de vuelta al cliente.

El enlace entre el cliente y el servidor es dinámico y se establece en cada invocación lo que permite a un SOO como este dar un mejor soporte a características como la movilidad de objetos.

El paso de mensajes no sólo asegura la separación de servicios ofrecidos por un objeto y su representación interna, sino que también proporciona independencia de su localización física. Así, el paso de mensajes subsume tanto invocación local como remota (el hecho de que origen y destino estén en distintas máquinas es transparente al programador), así como la posible inaccesibilidad de un objeto.

3.1.3 Integración de la Concurrencia en el modelo

El modelo de objetos activo es un modo natural de integrar concurrencia en los objetos. Si el modelo de objetos es activo, los objetos se definen como entidades activas que encapsulan su computación gestionándola como deseen.

Cada objeto es capaz de recibir mensajes de otros objetos a la vez que procesa algunos métodos ya invocados, de forma concurrente. A la hora de servir un método, puede establecer criterios para aceptar o retrasar su ejecución, en base a su propio estado interno.

Puede servir peticiones de forma síncrona o asíncrona y el propio mecanismo de paso de mensajes es un método primario de sincronización.

3.1.4 Entorno

El entorno de computación basado en un modelo de objetos activo se puede describir conceptualmente como un mundo de objetos autónomos donde cada objeto puede cooperar con otros para llevar a cabo una tarea común.

Estos objetos actúan e interaccionan concurrentemente mediante el envío de mensajes de tal forma que la ejecución de programas sigue el flujo dinámico implicado por los mensajes, sin imponer una sobrecarga de sincronización innecesaria[AK98]. La recepción de mensajes inicia la ejecución del método especificado con los argumentos indicados.

Por ejemplo, la arquitectura Cognac[MHM+95], precursora de Muse[YTM+91] y Apertos[Yok92], propone un modelo similar.

En Cognac, un objeto se compone de datos, módulo de comunicación, cola de mensajes y su contexto de ejecución. Cada mensaje que llegue al objeto, concretamente al módulo de comunicación, se encola en la cola de mensajes. Cada objeto tiene además la posibilidad de seleccionar un mensaje de los de la cola para ejecutarlo a continuación.

Como la comunicación es vía paso de mensajes, el objeto puede estar ejecutando un método concurrentemente con la recepción de otros mensajes. Y, por supuesto, dos objetos pueden estar ejecutándose concurrentemente.

En Cognac se define un **cluster de objetos** (del inglés *object cluster*) como el dominio de ejecución de un objeto e incluye varios meta-objetos: planificador, cola de mensajes, cola de mensajes y estado del objeto. El planificador determina la planificación de los objetos del cluster, la cola de mensajes encola los mensajes y el estado del objeto mantiene el estado de ejecución de cada objeto del cluster.

4 Comparación de ambos modelos

A continuación se establecen las ventajas e inconvenientes de ambos modelos para, al final realizar una propuesta de modelo para el SIOO propuesto.

4.1 Modelo de Objetos Pasivo

4.1.1 Ventajas

Proximidad conceptual a los entornos tradicionales

El modelo de objetos pasivo se basa en la existencia de dos abstracciones tradicionales, los procesos y los objetos, que en un entorno OO pueden estar implementadas siguiendo la tecnología de orientación a objetos.

De la existencia de una abstracción de proceso, se deriva una ventaja innegable de este modelo como es la proximidad a los entornos de computación tradicionales basados, precisamente, en dicha abstracción.

Es muy sencillo tanto diseñar como implantar un mecanismo de computación basándose en esas dos abstracciones. Una que encapsule un recurso ofreciendo métodos para su acceso y manipulación y otra que sea capaz de ejecutar dichos métodos modificando el estado del objeto que los ofrezca.

Familiaridad

Una ventaja innegable es la familiaridad del modelo, ya que cualquier lenguaje de programación convencional sirve como lenguaje base.

Gran cantidad de trabajos existentes

Una de las ventajas fundamentales es la gran cantidad de trabajos existentes sobre este modelo de objetos.

Muchos de los primeros intentos de construir un sistema orientado a objetos se basaban en una aproximación de este tipo. Como ejemplos se pueden citar Emerald[Hut87] como sistema orientado a objetos, Choices[CIM+93, LTS+96] como sistema operativo OO, ..

Relativa eficiencia

En principio se puede hablar de una eficiencia relativa, derivada fundamentalmente de los mecanismos de interacción entre objetos.

Los estudios realizados han dado como resultado un mejor rendimiento de sistemas de este tipo, fundamentalmente por las optimizaciones en la implantación (la invocación en este modelo es una correspondencia entre espacios de direcciones, mientras que en el caso del paso de mensajes, hay que construir el mensaje, empaquetar los parámetros, etc.).

4.1.2 Inconvenientes del modelo pasivo

Abstracciones no adecuadas ® Falta de Uniformidad y de Homogeneidad

Las abstracciones que ofrece implican un modelo no uniforme en el que existen objetos con distinta semántica en el sistema.

Así, los objetos pasivos se comunican a través del paso de mensajes o invocación a métodos mientras los objetos proceso, se comunican a través de la ejecución de métodos en objetos pasivos compartidos. No se utiliza pues el mismo paradigma de comunicación en todos los objetos del sistema.

La homogeneidad entre objetos se pierde, ya que los hilos son objetos porque son instancias de las clases pero, conceptualmente no tienen el mismo significado que los otros objetos.

Coste derivado de la falta de adecuación de las abstracciones

En este modelo, el proceso es la unidad de encapsulación para la ejecución de programas. Para poder ejecutar un método en un objeto, debe formar parte del espacio de direcciones del proceso.

De esto se deriva un coste extra al tener que establecer la correspondencia de los objetos en el espacio de direcciones de un proceso para, posteriormente, deshacer esta correspondencia y establecer otra con otro proceso[CC91].

Por otro lado, la introducción de asincronismo en el paso de mensajes, da al traste con la eficiencia que se consigue.

Violación del Principio de Encapsulacion

Dado que los procesos se ejecutan independientemente, es posible que ocurran problemas de exclusión mutua si dos o más hilos ejecutan simultáneamente el mismo método, sin sincronizar su acceso, violando la encapsulación de los objetos pasivos.

De la misma forma, la encapsulación de los clientes se viola si se les impone a ellos la responsabilidad de sincronizar explícitamente las solicitudes en objetos pasivos compartidos por muchos procesos [Nie93].

Protección

El sistema de seguridad propuesto[DAT+98] se basa en comprobar capacidades en el paso de mensajes.

La comunicación y ejecución de métodos, basada en la invocación de procedimientos complica el mecanismo de protección.

Problemas para la distribución y la persistencia

Finalmente, no es fácil extender este modelo a un entorno distribuido sin emplear alguna forma de paso de mensajes oculto.

Los mecanismos de comunicación y persistencia, difieren en el caso de una arquitectura distribuida, frente a una arquitectura centralizada. Sin embargo, se pretende un mecanismo que vuelva transparente la arquitectura subyacente.

En el caso distribuido, el modelo de objetos pasivo implica la construcción de estructuras de computación distribuidas que plantean problemas de identificación. Así,

es necesario transformar un proceso en un proceso distribuido de forma que, en cualquier momento, sea factible averiguar la secuencia de ejecución de métodos, tanto en una máquina como en varias.

Además, la necesidad de un mecanismo asíncrono en el que se crea un nuevo proceso para ejecutar el método, implica trabajar con dos modelos distintos que deben ser controlados simultáneamente.

4.2 El modelo de objetos activo

4.2.1 Ventajas

Soporte a una abstracción única y homogénea

La ventaja fundamental de la adopción de un modelo de objetos activo es la abstracción única (**objetos homogéneos**, **economía de conceptos**)

Todo es un objeto y no hay distinción entre objetos de usuario o de sistema ni entre unos objetos que tienen capacidades de las que otros carecen.

Frente a esto, en un modelo de objetos pasivo, el sistema se estructura como un conjunto de objetos pasivos y una clase especial de objeto activo que representa al proceso.

Uniformidad en la interacción de objetos

Todos los objetos se comunican mediante el paso de mensajes, no existe otro mecanismo de comunicación definido.

Facilita la sincronización

Una ventaja de este modelo es que facilita la sincronización[GK97b]. Es posible definir políticas de sincronización inherentes al modelo de objetos que hagan innecesaria la adición de gran parte del código de sincronización que plantea problemas con la herencia.

Fácilmente distribuible y persistente

Internet es una tendencia importante que apunta a la convergencia de computación paralela y distribuida. Con las mejoras en tecnología de redes y comunicación, se puede ver Internet como una máquina distribuida lo que pone de manifiesto la necesidad de transparencia en la distribución, movilidad, etc.[VA98, AK98]

En un modelo de objetos activo, se permite la comunicación de cualesquiera dos objetos de forma homogénea además de no ser necesaria una sincronización explícita dado que el paso de mensajes engloba tanto comunicación como sincronización[Nie89].

Ofrece una única abstracción, el objeto, para modelar los sistemas. Estos objetos son auto-contenidos, es decir, incluyen los datos, los métodos y la computación y, por tanto, son quienes la gestionan.

Los objetos realizan acciones ante la llegada de mensajes. Por tanto, el mismo mecanismo que sirve para activar un método también sirve para sincronizar ejecuciones.

Es un modelo muy adecuado para la implantación de mecanismos de persistencia y distribución completos y robustos ya que proporciona una visión única del objeto haciendo transparente la computación a dichos mecanismos.

De esta ventaja se derivan otras como son la Uniformidad para la distribución y para la Persistencia, facilita los mecanismos de migración y el Equilibrio de carga.

4.2.2 Inconvenientes

Eficiencia

El principal inconveniente que se le achaca a este modelo es la eficiencia. Sin embargo, factores como la distribución, en el que la eficiencia entre ambos modelos se iguala, y la necesidad de establecer continuamente correspondencias entre espacios de direcciones en el modelo pasivo, hacen que la sobrecarga del modelo se vea considerablemente mermada. Además, muchos autores están trabajando en optimizaciones del modelo que limen todavía más la diferencia de rendimiento con el modelo pasivo.

4.3 Propuesta para el SIOO

Modelo Activo

Se opta por un **modelo activo** que no ofrezca una visión dual del mundo de objetos sino una visión uniforme. Todos los objetos son activos e iguales presentando un sistema conceptualmente más sencillo para el usuario ya que no hay entidades dispares e independientes que sirvan para almacenar y ejecutar cosas.

Con este modelo, los objetos activos autocontienen toda la información referida a ellos, no sólo sus datos y métodos sino también la referida a la computación. Con esto, son los objetos los que, previo examen de esta información, pueden decidir su comportamiento en cada momento. Así, deciden cuándo ejecutar una invocación, cuándo pararla, cuándo reanudarla, etc. Este modelo ofrece grandes ventajas para la distribución y la persistencia.

Además, dado que cada objeto encapsula toda su computación, es sencillo implementar mecanismos de control de la concurrencia que estarían ligados a cada objeto.

Objeto Multihilo

Cada objeto, por su parte, es capaz de ejecutar varios métodos simultáneamente, siempre y cuando, las restricciones de sincronización establecidas lo permitan.

Esto favorece el rendimiento, fundamentalmente, en ambientes multiprocesadores y distribuidos.

Capítulo XI

Arquitectura del Sistema Integral

La problemática planteada en esta tesis se puede clasificar en dos vertientes. Por un lado, tal y como se adelantó en el Capítulo V, la construcción de un SIOO que adopte el paradigma de OO de manera uniforme en todas sus capas, de forma que solucione los problemas planteados en los primeros capítulos de esta tesis.

Por otro lado, la definición de un modelo de objetos para el SIOO que introduzca la concurrencia en el modelo.

En los capítulos precedentes se mostró cómo introducir la concurrencia es una tarea compleja que exige modificar la forma usual de construcción del sistema de computación. En lugar de ofrecer abstracciones explícitas para la concurrencia (procesos, hilos) se encapsula la ejecución en los objetos.

En el presente capítulo se vuelve nuevamente sobre el SIOO, para determinar la arquitectura de las capas más básicas como son la MAOO y el SOOO.

1 Integración de Máquina Abstracta y Sistema Operativo para formar un Sistema Integral

La arquitectura del sistema integral orientado a objetos propuesta se compone de una **máquina abstracta** y un **sistema operativo**.

La máquina abstracta ofrece el soporte básico a los objetos y los mecanismos básicos para su gestión y el sistema operativo modifica y extiende el comportamiento de la máquina abstracta. Juntos ofrecen un sistema de computación flexible orientado a objetos.

Dado que la abstracción única ofrecida por el sistema integral son los objetos, el sistema operativo debe proporcionar a los objetos un entorno de computación en el que puedan ejecutarse, comunicarse y sincronizarse de forma segura y totalmente independiente de la localización física del objeto (transparencia), sin romper el paradigma de OO. Es decir, el sistema operativo debe ofrecer su funcionalidad como objetos.

El entorno de ejecución debe ser flexible, es decir, cada objeto debería tener la posibilidad de diseñar su propio entorno de computación de acuerdo a sus requisitos específicos.

2 ¿Cómo ofrecer la funcionalidad del sistema operativo?

Para diseñar el entorno de computación del SOOO, es necesario no olvidar que se sustenta sobre una máquina abstracta orientada a objetos que ofrece no sólo el soporte básico a los objetos sino además un conjunto de clases que ofrecen la funcionalidad básica del sistema.

Existen fundamentalmente tres posibilidades para integrar Máquina Abstracta y Sistema Operativo, de forma que el resultado sea un entorno de computación concurrente OO: ampliación de la máquina, ampliación de la semántica del conjunto de clases básicas y colaboración con la máquina por parte del usuario utilizando algún mecanismo como la reflectividad.

3 Implementación ad-hoc en la propia máquina abstracta

Una posibilidad para añadir una característica al sistema integral, como puede ser la introducción de concurrencia en el modelo, sería la modificación del funcionamiento o la estructura de la máquina para proporcionarla.

Estas modificaciones suponen cambios en la interfaz de alto nivel y, por tanto, conllevan modificaciones en la implantación interna de la máquina para soportar ese cambio. Aquellas modificaciones en la implantación por cuestiones de rendimiento u otras, que no provoquen cambios en la interfaz son totalmente transparentes y no se consideran aquí.

Un caso claro, relacionado con la computación es la posibilidad de ampliar el soporte que la máquina abstracta ofrece a la existencia y gestión de los objetos, para encargarse de ofrecer las políticas de comunicación, planificación y sincronización de los objetos.

3.1 Existencia de Objetos

Como ya se mencionó en el Capítulo V, el soporte de la abstracción de objeto se integra en la capa más básica del SIOO, en este caso, la máquina abstracta.

Así, esta ofrece soporte completo a la existencia de objetos, de tal forma que las capas superiores, incluida el sistema operativo, heredan de ella una abstracción de objeto que, tal y como se definió en el capítulo X, seguirá un modelo activo.

3.2 Comunicación

Como mecanismo de comunicación básico entre objetos, se adopta el paso de mensajes.

Una primera aproximación a la implantación del mecanismo de paso de mensajes en el SIOO es la integración total del mecanismo en la MAOO como paso de mensajes síncrono. De esta forma, la máquina se encarga de llevar a cabo todos los pasos para comenzar a ejecutar el método solicitado en el objeto requerido.

La integración completa del mecanismo de comunicación a tan bajo nivel, supone que, la introducción de un modelo de paso de mensajes asíncrono implica la necesaria modificación de la máquina y de su interfaz de alto nivel, al ser necesario añadir una nueva instrucción al ensamblador de la máquina.

3.3 Concurrencia

Si se adopta un modelo de paso de mensajes síncrono, surge la necesidad de introducir de alguna forma la concurrencia en el sistema.

Una posible modificación de la máquina sería la introducción de cierto paralelismo catalogando los métodos como **mensajes** (messages) en cuyo caso se crea un nuevo hilo para la ejecución de ese método que continúa en paralelo con el método que realizó la llamada, o **métodos** (methods), en cuyo caso se lleva a cabo una invocación síncrona tradicional.

Esto supone la modificación, no sólo internamente de la máquina, sino también de la interfaz de alto nivel de la misma, ya que supone la adición de una instrucción declarativa que permita establecer la diferencia entre los métodos.

3.4 Sincronización

En las primeras versiones de la máquina propuesta[Izq96], esta ofrece soporte para la concurrencia utilizando la interfaz de la máquina en dos sentidos.

Métodos y Mensajes

En primer lugar, el propio mecanismo de paso de mensajes puede servir para establecer un primer nivel de sincronización, para lo que se introducen dos instrucciones declarativas (method, message) que califiquen los métodos de una clase.

Modificadores y Selectores

En segundo lugar, la sincronización intra-objetos la lleva a cabo con instrucciones declarativas tipo concurrent que diferenciaban métodos selectores de métodos concurrentes. Ante la invocación de alguno de ellos, la máquina abstracta decide ejecutar o no el método en función del estado del objeto.

La introducción de este mecanismo de control de la concurrencia, basado en instrucciones declarativas, supone, al igual que en el caso anterior, la modificación interna y de la interfaz de la máquina.

3.5 Planificación

Por el contrario, el caso de la planificación, es uno de aquellos en los que, la adición de alguna característica a la máquina, no supone modificación de la interfaz y, por tanto, no se le da el mismo tratamiento.

La forma más sencilla de añadir la planificación al sistema es integrarla dentro de la máquina abstracta. De esta forma, la planificación se encuentra codificada internamente dentro de la MAOO y se realiza de forma completamente transparente al usuario.

Sin embargo, implementar el planificador como un objeto interno (C++) de la máquina supone que, ante la necesidad de especializar el planificador de acuerdo a las necesidades de las aplicaciones, será necesario "meter mano" a la propia máquina.

Sin embargo, esto no será visible al exterior, excepto por una mejora en el rendimiento.

3.6 Ventajas e Inconvenientes

3.6.1 Ventajas

La ventaja fundamental de esta aproximación es la **facilidad de implantación**. Contando con el código fuente de la máquina, se pueden hacer modificaciones a voluntad sin necesitar más conocimientos que el lenguaje de desarrollo utilizado.

3.6.2 Inconvenientes

Sin embargo, frente a la ventaja anterior, se presentan varios inconvenientes.

- **Falta de adaptabilidad**. Las aplicaciones que se ejecuten sobre la máquina abstracta pueden utilizar los mecanismos que ofrece, pero sin posibilidad de cambiar ni adaptarlos a requerimientos concretos. Por ejemplo, si un objeto quiere enviar un mensaje a otro, es posible que desee que si el objeto destino no está en su máquina, llevar a cabo una serie de acciones especiales. Si la operación de invocación de métodos está implementada a bajo nivel en la máquina abstracta, no será posible.
- **Poco flexible**. La falta de separación de política y mecanismos provoca una ausencia total de flexibilidad. Los dilemas de correspondencia se mantienen.

4 Adición y/o modificación de las clases básicas de la máquina creando un conjunto de objetos distinguidos o especiales

Existen un conjunto de clases básicas (Object, Integer, ...) implementadas en la máquina en el lenguaje de desarrollo utilizado para implementarla a ella (C++ o cualquier otro).

Cualquier aplicación se construye basándose, no sólo en las instrucciones ofrecidas por la máquina, sino utilizando las clases que ofrece, instanciando objetos de dichas clases, especializándolas o añadiendo nuevas clases.

Para añadir nuevas características o modificar el comportamiento del SIOO, una posibilidad, además de la anterior, podría ser la modificación de la funcionalidad de estas clases mediante la adición de nuevos atributos y/o métodos lo que conlleva un cambio en el funcionamiento de la máquina de cara al exterior o la modificación de su comportamiento.

Un ejemplo es la posibilidad de modificar la semántica del objeto para ofrecer un modelo de objetos activo. La máquina abstracta tiene como raíz de la jerarquía que expone al exterior la clase objeto, clase de la que cualquier otra clase, definida en la

máquina o por el usuario, debe heredar, heredando por tanto, la semántica definida por ella.

Si esa clase se implanta dotando al objeto de características relacionadas con la computación, tal como se vio en el capítulo X, la máquina abstracta da soporte a una abstracción de objeto activo y cualquier objeto que se cree en el sistema también seguirá este modelo, lográndose la ansiada homogeneidad.

4.1 Ventajas e Inconvenientes

4.1.1 Ventajas

Facilidad de Implantación

La ventaja fundamental de esta aproximación es, al igual que en el caso anterior, la **facilidad de implantación**. Y, frente a ella, cuenta además con una ventaja adicional como es una mayor transparencia al no modificarse el juego de instrucciones de la máquina.

4.1.2 Problemas:

Sin embargo, adolece de muchos de los inconvenientes ya citados para la anterior aproximación expuesta.

Falta de Flexibilidad

Dotar al objeto de un modelo activo puede hacerse de forma que el objeto permita la ejecución simultánea de varios de sus métodos, en cuyo caso, deberá proveer una interfaz para especificar la sincronización entre ellos, o bien, puede definirse un objeto que serialice sus actividades, ejecutando las peticiones una a una. Es evidente que el entorno definido cambia completamente.

La modificación de la implantación de la clase objeto se modificando internamente la máquina abstracta. Debido a esto, las posibles implantaciones de objeto son fijas, definidas por el desarrollador de la máquina e inamovibles.

Utilidad limitada

No todas las características se pueden integrar en el SIOO modificando el conjunto de clases básicas. Por ejemplo, nuevos mecanismos de comunicación o nuevas políticas de planificación, no dependen de la implantación de los objetos.

5 Colaboración con el funcionamiento de la máquina. Reflectividad

Una tercera posibilidad es utilizar la máquina abstracta para ofrecer el soporte básico a la existencia de objetos, así como los mecanismos básicos para construir un entorno de ejecución para los mismos, a la vez que se permite que sus mecanismos puedan ser configurados por los propios objetos del sistema operativo que son objetos no diferentes de cualquier objeto de usuario normal.

Así, los objetos del sistema operativo modificarían la funcionalidad de la máquina abstracta extendiendo su comportamiento mediante objetos normales.

Para ello, es necesario que la máquina esté construida de tal forma que permita que un objeto externo sustituya a un objeto propio para ofrecer la misma funcionalidad de una forma distinta manteniendo de esta forma la **uniformidad en torno a la OO.**

El problema es cómo realizar la comunicación entre la máquina y los objetos de usuario sin romper la uniformidad en la OO. La solución es dotar a la máquina de una arquitectura reflectiva OO [Mae87], tal y como se anticipó en el Capítulo II.

La reflectividad hace que la máquina sea expuesta a los objetos de usuario en forma de un conjunto de objetos normales, en la que los objetos de la máquina no se diferencian de los objetos normales. Así, los objetos de la máquina pueden usar los objetos normales, y viceversa, usando el mismo marco de la OO común en el sistema.

El sistema se puede ver como un **conjunto de objetos homogéneos** que interactúan entre sí, compartiendo las mismas características (el mismo modelo de objetos). Algunos de ellos serán objetos de la máquina, otros objetos normales de usuario. De estos últimos, algunos implementarán funcionalidad del sistema operativo. Sin embargo, el origen de un objeto es transparente para todos los demás y para sí mismo.

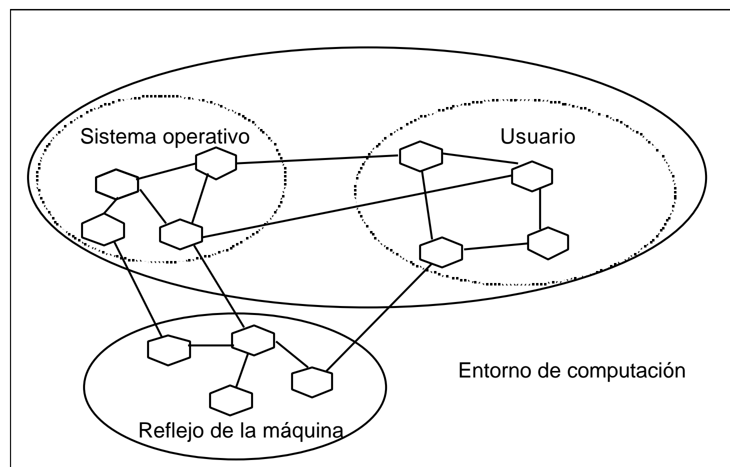


Figura 11.1 Espacio de objetos homogéneo formado por la unificación de los objetos de la máquina con los del sistema operativo y los del usuario.

La reflectividad está reconocida como una técnica importante para lograr flexibilidad (extensibilidad, adaptabilidad, etc.) en los núcleos de los sistemas operativos [Cah96a, GW96]. Cuando se utiliza en un sistema integral orientado a objetos contribuye a la uniformidad, como se ve. Debido a la importancia de la flexibilidad para el sistema integral el siguiente capítulo se dedica a un estudio más profundo de la reflectividad para hacer una selección de qué tipo de reflectividad y qué alternativas existen para implantarla en la máquina abstracta.

Capítulo XII

Estudio de la Reflectividad como Arquitectura Integradora

Dado que la reflectividad se ha manifestado como la arquitectura más adecuada para estructurar el SIOO, concretamente, para organizar el modo en que MAOO y SOOO cooperan, se realiza aquí un estudio pormenorizado de la misma.

En el capítulo XIV se aplican los conceptos que aquí se exponen al caso concreto de la máquina abstracta y el sistema operativo propuestos en esta tesis.

1 Reflexión: Modificación de la estructura o el comportamiento de un sistema

1.1 Reflectividad

En términos humanos, **reflexión** (del inglés *reflection*) se refiere normalmente al acto de pensar acerca de las propias ideas, acciones y experiencias.

En el campo de los sistemas de computación, la reflectividad apareció en primer lugar en el campo de la Inteligencia Artificial aunque se propagó rápidamente a otros campos como los lenguajes de programación, donde destaca el trabajo [Smi82], y las tecnologías Orientadas a Objeto, donde fue introducido por Pattie Maes en [Mae87].

Existen varias definiciones de **reflectividad** aunque quizá la más extendida, con algunas modificaciones, encaje con la siguiente[BGW93]:

"Reflectividad es la habilidad de un programa de manipular, como si de datos se tratasen, la representación del propio estado del programa durante su ejecución".

En esta manipulación existen dos aspectos fundamentales: **introspección** e **intercesión**. La introspección es la habilidad de un programa de observar y razonar acerca de su propio estado. La intercesión es la habilidad de un programa de modificar su propio estado de ejecución o alterar su propia interpretación[BGW93, MJD96].

Ambos aspectos requieren un mecanismo que codifique el estado de ejecución como datos. La **exposición**, del inglés *reification*, es proporcionar tal codificación.

En un sistema de computación, la reflectividad es, por tanto, "*el proceso de razonar acerca de y actuar sobre el propio sistema*" [Mae88]. Esto implica no sólo poder conocer la estructura interna de parte o todo el sistema, sino además, ser capaz de modificar alguna o todas sus partes.

Formalmente, la reflectividad fue introducida en la computación por Brian Smith [Smi82] que la definió como "*la habilidad de una entidad de representar, operar o tratar de cualquier modo consigo misma de la misma forma que representa, opera y/o trata con entidades del dominio del problema*". Dicho de otra manera, la reflexión es la habilidad de un sistema de manipular, de la misma forma que manipula los elementos que representan el problema del mundo real, elementos que representan el estado del sistema durante su propia ejecución.

1.2 Modelo del Sistema

El principal motivo de añadir reflexión a los sistemas de computación es **hacer que la computación del nivel base se lleve a cabo de forma adecuada para la resolución de un problema del mundo real**.

Para ello, es necesario hacer posible la **modificación de la organización interna del sistema de computación**[YW89] facilitando el acceso a esta estructura interna como si fuesen datos, ofreciendo los mecanismos necesarios para razonar acerca de su estado actual y posibilitando la modificación del mismo para adecuarse a las condiciones actuales.

El proceso de razonar y actuar sobre el sistema implica, por tanto, el **acceso y modificación de parte o de todo el sistema**.

Debido a esto, uno de los problemas que se le puede achacar es la posibilidad de manipulación directa y sin control de la estructura interna del sistema lo que derivaría rápidamente en el caos. Sin embargo, los mecanismos de reflectividad no permiten la manipulación directa del sistema de computación, sino que se ofrece un **modelo o representación restringida del sistema** que sí está permitido cambiar y manipular[YW89, Riv88].

El modelo del sistema debe satisfacer varios requisitos para exponer completamente los aspectos del sistema elegidos y permitir la modificación de los mismos de forma efectiva y segura[YW89].

- **Abstracción adecuada del sistema**. El nivel de abstracción que el modelo aplica sobre el sistema debe ser adecuado con relación al uso que se pretenda hacer de la reflectividad.
- **Adecuado para el Razonamiento y la Manipulación**. El modelo debe representar al sistema de forma que sea sencillo razonar sobre él y su estado, así como la manipulación del mismo.
- **Relación de Causalidad. Conexión Causa-Efecto.** El modelo debe estar conectado con el sistema mediante una relación de causalidad de tal forma que los cambios efectuados sobre el modelo afecten al comportamiento posterior del sistema.

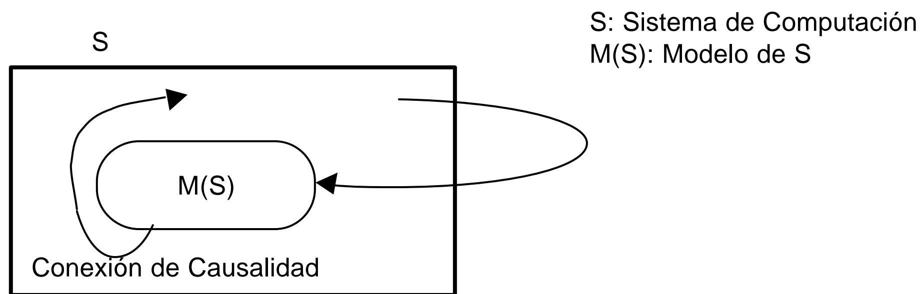


Figura 12.1 Sistema Reflectivo. El Modelo y la Relación de Causalidad

1.3 Actividades básicas en un sistema reflectivo: Exposición y Reflexión

En el proceso de reflexión se identifican dos actividades básicas: **exposición**, del inglés, *reification* y **reflexión**, del inglés *reflection*[Mae87].

1.3.1 Exposición

Los dos aspectos básicos de la reflectividad, **introspección** e **intercesión**[BGW93, Mae87], requieren un mecanismo que codifique el estado de ejecución de ciertos aspectos del sistema como datos accesibles, creando un modelo del sistema, a la vez que ofrecen mecanismos adecuados para la manipulación de tales aspectos de sí mismo[Mae87].

La **exposición**, es la habilidad de proporcionar tal representación, transformando y haciendo accesibles aquellos aspectos de la implantación que, normalmente, están ocultos[BGW93, CS98].

1.3.2 Reflexión

La **reflexión** es el acto de modificar algunos aspectos del sistema por el sistema en sí **gracias a la actuación sobre el modelo ofrecido**. De esta forma, cualquier cambio en el modelo, en los aspectos expuestos, se refleja en el comportamiento del sistema [CS98, YW89].

Existe pues una correspondencia entre los aspectos reflejados y su representación, de tal forma que cualquier modificación en la representación modifica el aspecto representado y viceversa[YW89].

2 Arquitectura reflectiva

Las arquitecturas reflectivas proporcionan, fundamentalmente, un **nuevo paradigma de computación**. Tal es así que, en una arquitectura reflectiva, un sistema de computación se ve desdoblado en una **parte objeto**, que exhibe una **computación objeto** y cuya tarea es solucionar un problema en el dominio externo y retornar los resultados, y una **parte reflectiva**, que exhibe una **computación reflectiva** y cuya tarea es solucionar problemas y retornar información acerca del propio sistema y su computación[Mae88].

La computación reflectiva no contribuye directamente a solucionar problemas en el dominio del problema, sino que contribuye a la organización interna del sistema o a su interfaz al mundo externo. Su propósito es garantizar el funcionamiento adecuado de un sistema de computación.

Una arquitectura reflectiva debe proporciona un medio para implantar la computación reflectiva de **manera modular**, lo que hace al sistema más manejable, comprensible y fácil de modifcar.

Es, pues, común pensar en un sistema reflectivo estructurado o compuesto, desde un punto de vista lógico, por dos o más niveles, que constituyen una **torre reflectiva**[WF88]. Cada uno de estos niveles sirven como **nivel base** o dominio del problema para el nivel superior y como **meta-nivel** para el nivel inferior.

La parte base soluciona el problema externo mientras la parte reflectiva mantiene información y determina la implantación de la parte base.

2.1 Definición de una Arquitectura Reflectiva

En este nuevo paradigma, para definir un entorno de computación, es necesario determinar ciertos parámetros como los que siguen[Fer89].

Qué forma la parte reflectiva

Para llevar a cabo la reflectividad en un sistema, es necesario definir qué aspectos se quieren reflejar, es decir, qué entidades y/o características deben exponerse.

Estas entidades o aspectos del sistema de computación en sí, serán aquellos sobre los que actúa la parte reflectiva de la arquitectura para lo que deben representarse o transformarse en algo que pueda ser manipulado en el meta-nivel, en oposición a aquello sobre lo que actúa el nivel base.

En esta tesis se proponen tres aspectos de la máquina y el sistema operativo como candidatos a ser expuestos, a saber, la comunicación, la planificación y el control de la concurrencia.

Representación del sistema

Además de decidir qué se expone, para definir una arquitectura reflectiva, es necesario definir la representación del sistema dentro del sistema. Hay, al menos, dos aproximaciones para construir la auto-representación de un sistema de computación:

- Asumir la existencia de un conjunto de datos que representan al sistema.
- Introducir la auto-representación de cada entidad de forma individual en el sistema, que será la aproximación escogida en esta tesis, así como en otros trabajos como [WY88].

A la vez que es necesario representar el sistema, también es necesario mantener la **relación causa-efecto** entre el modelo o representación del sistema y el sistema en sí, de forma que las acciones realizadas sobre el modelo, tengan su reflejo en el comportamiento del propio sistema.

En esta tesis se propone llevar a cabo la relación causa-efecto, mediante la representación de partes de la propia máquina abstracta y el sistema operativo como objetos, de forma que, al modificar uno de esos objetos sustituyéndolo por otro, el

comportamiento de la propia máquina y el sistema operativo se ve inevitablemente modificado.

Flujo de control entre el nivel-base y el meta-nivel

Por último, es imprescindible determinar cómo se activa la meta-computación y en qué momento el nivel base vuelve a tomar el control.

La propuesta de esta tesis, a tenor de lo visto en los dos apartados anteriores, no podía ser otra. Debido a la representación de las características expuestas como objetos, se propone utilizar el mismo mecanismo de comunicación para comunicar objetos que para transmitir el control entre el nivel base y el meta-nivel.

Esta transmisión se produce, invariablemente, cuando la máquina abstracta no implementa *ad-hoc*, alguna característica solicitada por el nivel base, sino que esta característica se implementa como un objeto. En este caso, se envía un mensaje a tal objeto que, una vez terminado, retornará el control al nivel base para proseguir su ejecución.

Al igual que en los apartados anteriores, se emplaza al lector a los próximos capítulos para una descripción más detallada de este punto.

2.2 Aplicación de la Reflectividad en sistemas Orientados a Objetos

La reflectividad encaja especialmente bien con los principios promovidos por las tecnologías Orientadas a Objetos[BG96] que insisten en la encapsulación en los niveles y la modularidad en los efectos.

La **parte reflectiva** de un sistema Orientado a Objetos se puede diseminar a través de una **constelación de objetos**, cada uno de los cuales expone ciertos aspectos de la estructura subyacente[Foo90].

La bondad de esta idea es fundamental para la adopción de una arquitectura reflectiva como mecanismo de estructuración del sistema operativo y la máquina abstracta OO, punto fundamental en esta tesis.

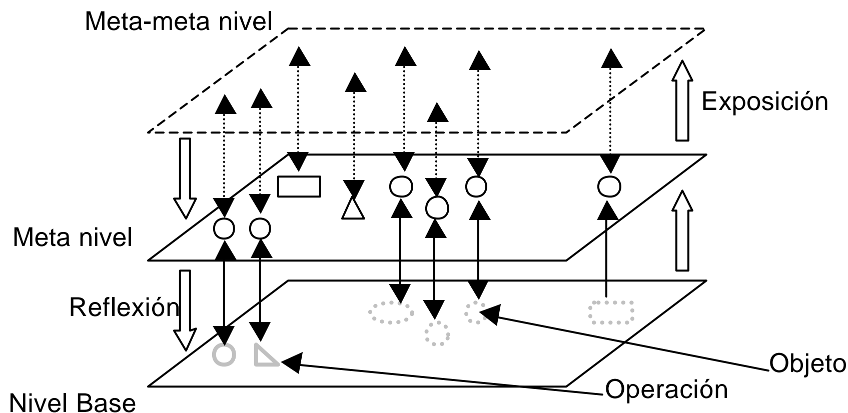


Figura 12.2 Elementos involucrados en el proceso de reflexión: meta niveles, exposición y reflejo

2.2.1 Representación del modelo del sistema como un conjunto de objetos

Es posible organizar el control del comportamiento de un sistema de computación OO, su **meta-interfaz**, a través de un conjunto de objetos que pueden representar varias características del contexto de ejecución tales como representación, ejecución, comunicación o localización, entre otras[BG96, WY88, Fer89].

Especializando estos objetos, a los que pronto se dará el nombre de **meta-objetos**, es posible extender y modificar el contexto específico de ejecución de objetos de la aplicación.

De esta forma se adapta el entorno de ejecución a los requerimientos puntuales de cada objeto o conjunto de objetos de una aplicación, eliminando los dilemas y conflictos definidos en el capítulo II.

La reflectividad también puede ayudar a expresar y controlar la gestión de recursos, como la planificación.

2.2.2 Uniformidad

La representación del modelo del sistema como una colección de objetos hace que la utilización de las capacidades reflectivas por parte de una aplicación se haga por medio del paso de mensajes a los objetos que representan el modelo manteniendo la uniformidad conceptual alrededor del concepto objeto[YW89, Fer89]

2.2.3 Parámetros de una arquitectura reflectiva OO

Supuesto un entorno OO, la arquitecura reflectiva proporciona al menos tres primitivas, que son la traducción directa de los parámetros generales descritos en un apartado anterior dedicado a la Definición de una Arquitectura Reflectiva.

Invocación al meta-espacio

La primera primitiva permite a un objeto realizar peticiones a la parte reflectiva de la arquitectura. Dado que tanto el nivel base como el meta-nivel se representarán mediante objetos, este mecanismo de invocación puede ser, y de hecho es, simplemente, el mecanismo de invocación de métodos.

Reflejo de la meta-computación

La segunda primitiva habilita al meta-objeto a reflejar el resultado de la meta-computación a sus objetos. Puesto que los meta-objetos sustituyen el procesamiento normal, interpretando parte del comportamiento de los objetos, la modificación de uno de ellos se refleja en un cambio en el comportamiento del sistema.

Relación de Causalidad

La tercera primitiva mantiene el enlace causal entre objetos y meta-objetos. Las aplicaciones de usuario residen en el espacio de objetos y las funciones que tradicionalmente formaban parte del sistema operativo existen en el meta-espacio.

La asociación de meta-objetos a los objetos base mediante la ampliación de la implantación de la clase objeto, da soporte a esta relación.

Por ejemplo, planificación, comunicación entre objetos y otras son gestionadas en el meta-espacio.

3 Estructuración de los Sistemas Reflectivos

Aunque, como ya se ha dicho antes, una arquitectura reflectiva suele estar formada por varios niveles, para explicar el funcionamiento general es suficiente con dos, además de ser este el número de niveles escogido para la arquitectura reflectiva propuesta en esta tesis.

3.1 Nivel Base y Meta-Nivel

El primer nivel, **nivel base**, soluciona el problema externo describiendo la computación desde el punto de vista de la aplicación, es decir, la computación que el sistema lleva a cabo como entidades que reflejan el mundo real.

El **meta-nivel** mantiene información y describe cómo se lleva a cabo la computación en el nivel previo.

Las entidades que trabajan en el nivel base se denominan **entidades-base**, mientras que las entidades que trabajan en otros niveles(meta-niveles) se llaman **meta-entidades** [Caz98, YW89, Fer89, ZC96].

En un Sistema OO, un **objeto base** u **objeto del nivel base** es un objeto definido por la aplicación y que se refiere al ámbito del problema, mientras que un **objeto del meta-nivel** o **meta-objeto** es un objeto que representa información acerca de los objetos del nivel base que, de otra forma, sería implícita. Así, los meta-objetos –que reciben este nombre debido a que estarán representadas a su vez como un objeto o conjunto de objetos– conocen la estructura de los objetos del nivel base y determinan su comportamiento[Fer89].

Cada nivel está causalmente conectado con los niveles adyacentes, de tal forma que todos los niveles, excepto el primero y el último, son el nivel base para el nivel superior y el meta-nivel para su nivel inferior.

Las meta-entidades supervisan la actividad de las entidades base, para lo que tienen estructuras de datos que **exponen** las actividades y estructuras de las entidades que trabajan en el nivel inferior y sus acciones se reflejan en tales estructuras. Cualquier cambio a tales estructuras modifica el comportamiento de la entidad.

En definitiva, cada nivel es un modelo del nivel anterior de tal forma que, cualquier cambio a tales estructuras de datos modifica el comportamiento de la entidad base.

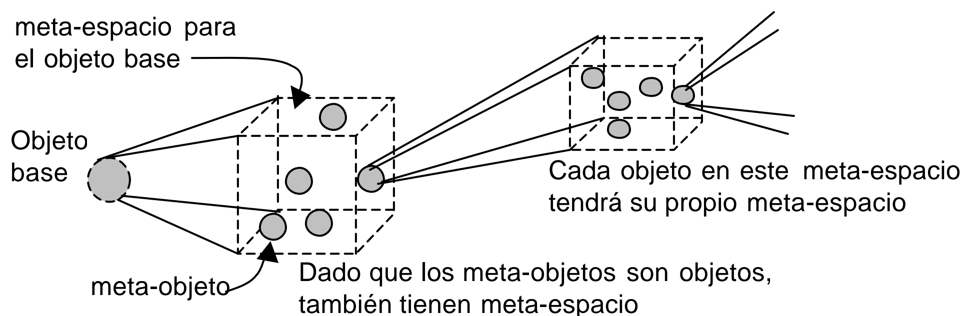


Figura 12.3 Relación objeto/meta-objeto

Se ha demostrado que el uso de meta-objetos para añadir reflectividad a los LOO toma en consideración la definición de nuevas características que no estaban presentes inicialmente en el sistema: herencia múltiple, herramientas de depurado, ...

3.2 Transferencia de control

Una parte fundamental de una arquitectura reflectiva es el mecanismo de transferencia de control desde el nivel-base al meta-nivel, es decir, bajo qué condiciones un sistema manipulará su auto-representación y qué clase de computación llevará a cabo el sistema cuando se den esas condiciones[Mae88].

3.2.1 Propiedades

El mecanismo de transferencia de control debe cumplir algunas propiedades:

- **Transparencia**. La transferencia de control debe ser transparente al objeto del nivel-base, es decir, el programador no debería necesitar considerar ninguna funcionalidad del meta-nivel a la hora de diseñar e implementar el objeto del nivel base.
- **Sincronismo**. La transferencia de control debería ser síncrona, lo que da al meta nivel la oportunidad de controlar la ejecución concurrente del nivel base. Diferentes mecanismos de sincronización pueden implantarse como programas al meta-nivel.

3.2.2 Arquitecturas Reflectivas, atendiendo a la transferencia de control

Se pueden identificar dos tipos de arquitecturas, dependiendo del momento y condiciones en que se transfiere el control del nivel-base al meta-nivel: **reflectividad explícita** y **reflectividad implícita**[Mae88].

Reflectividad Implícita

El **sistema activa sistemática y automáticamente la computación reflectiva**. Esto significa que existen agujeros en el proceso normal de interpretación que la computación reflectiva llenará, o lo que es lo mismo, si el proceso de interpretación consiste en la ejecución de una serie de tareas t_1, t_2, ..., t_n, existen uno o más i para los que la acción t_i no está definida por la máquina subyacente. Es decir, algún t_i es, necesariamente, un trozo de computación reflectiva definida por el programa que está siendo ejecutado (o interpretado).

En la arquitectura propuesta, esta será una de las formas en las que la meta-computación se lleve a cabo.

Reflectividad Explícita

La reflectividad explícita no ocurre automáticamente sino que **tiene lugar únicamente cuando la aplicación lo solicita**. De esta forma, la computación tiene lugar, normalmente, en el nivel de objetos. Siempre y cuando el programa solicite la computación reflectiva, el sistema, temporalmente, deja este nivel y pasa a realizar alguna tarea en el nivel reflectivo.

3.2.3 Manipulación del entorno

En cualquier caso, tanto si la reflectividad es explícita como si es implícita, los meta-objetos permiten la manipulación del entorno de computación de los objetos del nivel base. Esta manipulación se lleva a cabo en dos fases: **Paso al Meta-Nivel**, del inglés *shift-up action* o *lookup* y **Vuelta al Nivel-Base**, del inglés *shift-down action* o *apply*[Caz98].

Paso al Meta-Nivel

Una computación comienza con el flujo de computación en el nivel base. En un determinado momento, la entidad base puede ejecutar una tarea o secuencia de tareas que provoquen que el sistema solicite al meta-nivel la realización de alguna actividad.

Un ejemplo se produce cuando un objeto del nivel-base invoca un método de otra entidad del nivel base, o lo que es lo mismo, envía un mensaje.

La meta-entidad intercepta el envío y efectúa una meta-computación para el mensaje – **Reflexión implícita**. El flujo computacional va del nivel base al meta-nivel.

También es posible que la entidad del nivel base invoque un método del meta-nivel directamente –**Reflexión explícita**– para, por ejemplo, llevar a cabo una traza de determinadas características de la ejecución actual.

Vuelta al Nivel-Base

La meta-entidad tiene el control y, si es necesario, lleva a cabo alguna meta-computación. Cuando la completa, permite que la entidad base lleve a cabo la acción, el flujo de computación vuelve al nivel base (*shift-down action*).

En este caso, el flujo de control vuelve del meta-nivel al nivel base.

4 Tipos de Reflectividad

La integración de la reflectividad en los sistemas orientados a objetos puede hacerse de forma integral y sin fisuras y, sin embargo, existen algunos trabajos en tal dirección que han llevado a dos modelos diferentes de reflectividad[Fer89]: la **reflectividad estructural** y la **reflectividad del comportamiento** o **reflectividad computacional**, que no deben considerarse exclusivos.

El tipo de reflectividad soportado por el modelo especifica cuáles son los aspectos del sistema que se exponen y sobre los que, por tanto, se puede actuar [Fer89, Caz98].

4.1 Reflectividad Estructural

La **Reflectividad Estructural** se puede definir como la *habilidad de exponer tanto el programa que está siendo ejecutado en este momento como sus datos*[Caz98]. En caso de que se tratase de una Arquitectura Reflectiva OO, esta exposición se llevaría a cabo utilizando los conceptos de OO, como objeto o clases.

Este modelo tiene en cuenta la extensión de la parte estática de un sistema, es decir, permite exponer y manipular los aspectos estructurales del sistema de computación, como la herencia en un sistema OO, considerándolos como datos[Coi88] y ofrece mecanismos que permitan la manipulación de dicha representación.

Reflectividad Estructural aplicada a la OO

Los sistemas OO parecen ser uno de los campos en donde la reflectividad puede ser más fácilmente implantada, dada la naturaleza inherente de tales sistemas a distribuir en objetos los datos y procedimientos que representan el sistema.

La reflectividad estructural en sistemas OO, introducida por P. Cointe [Coi88], se refiere al uso reflectivo de clases y metaclases para implementar objetos.

En un modelo de clases puro, cada entidad es una instancia de una clase y las clases, a su vez, son instancias de otras clases, llamadas **Metaclases**. Este modelo tiene en cuenta la extensión de la parte estática de los LOO, es decir, de los aspectos estructurales de los objetos considerados como la implantación de un tipo abstracto de datos[BC87].

Muy extendida tanto en los lenguajes funcionales – Lisp –, como en los lenguajes lógicos – Prolog –, que tienen sentencias para manipular la representación del programa. Por ejemplo, Prolog representa los programas como estructuras y es posible acceder al functor, los argumentos, etc. Esta reflectividad se fundamenta en el hecho de que ambos son lenguajes interpretados.

En el caso de los lenguajes orientados a objetos, la mayoría son compilados y, por tanto, no hay representación de los mismos en tiempo de ejecución. Excepción hecha de Smalltalk que utiliza las clases para llevar a cabo la reflectividad estructural[BC89, FJ89].

Es adecuada cuando lo que pretendemos es dotar al sistema de mecanismos como la introspección y se aplica fundamentalmente a elementos estructurales del sistema como la herencia.

Como ejemplos conocidos y significativos de reflectividad estructural podemos citar la API Reflectiva de Java[Sun97], el RTTI de C++[Str92] o SOM[IBM96a, IBM96b].

4.2 Reflectividad del Comportamiento.

La **reflectividad del comportamiento**, introducida por P. Maes[Mae87] y B. Smith[Smi82], se refiere al comportamiento del sistema computacional y se puede definir como *la habilidad de un lenguaje – en general cualquier sistema – para proporcionar una exposición completa de su semántica y de los datos que utiliza para ejecutar el programa actual*.

La reflectividad del comportamiento manipula el comportamiento del sistema de computación en tiempo de ejecución e implica dotar a un programa P de la habilidad y los mecanismos necesarios para observar y modificar las estructuras de datos utilizadas para su propia ejecución [MJD96].

4.2.1 Superposición Reflectiva

Evidentemente, estas modificaciones pueden llevar a inconsistencias si las actualizaciones hechas por el código reflectivo, el que modifica el entorno en tiempo de ejecución, chocan con las actualizaciones hechas por el intérprete o sistema.

Este fenómeno, conocido como **Superposición Reflectiva**, del inglés **introspective overlap**, es estudiado por B.C.Smith [Smi82, Smi84], que establece una distinción entre

el intérprete – o sistema en tiempo de ejecución – P_1, que ejecuta el programa P, y el intérprete – o sistema en tiempo de ejecución – P_2 que ejecuta el código reflectivo.

4.2.2 Torre Reflectiva

De hecho, este argumento puede llevarse al extremo, permitiendo código introspectivo en el intérprete P_2, que necesitará otro intérprete P_3 y así sucesivamente, posibilitando un número potencialmente infinito de intérpretes.

Esta pila de intérpretes, denominada **torre reflectiva**, no necesitan estar basados en técnicas interpretativas, por lo que Smith y des Rivières [dRS84] propusieron el término alternativo de **Procesador Reflectivo**.

4.2.3 Reflectividad del Comportamiento en sistemas OO

La Reflectividad del Comportamiento en Sistemas Orientados a Objeto, se basa en el hecho de que cada objeto del nivel del problema, tiene asociado su propio objeto que representa la información implícita acerca de su existencia: su estructura y el mecanismo de gestión de mensajes[Mae87, Smi82].

En el siguiente apartado, se extenderá este concepto.

5 Modelos de Reflectividad

Existen diferentes modelos de reflectividad que se pueden aplicar a la reflectividad estructural y del comportamiento[1] aunque algunos modelos no son demasiado apropiados para algunos tipos. J. Ferber en [Fer89] establece una primera clasificación de los modelo de reflectividad.

En este trabajo, J.Ferber señala la existencia de dos aproximaciones reflectivas, principalmente. Tales aproximaciones son la **aproximación meta-objeto** y la **exposición de la comunicación**.

Aproximación Meta-Objeto

Esta primera aproximación, consiste en enlazar cada **entidad base**, también denominada **referente**, con una o más **meta-entidades**, también llamadas **meta-objetos** – dado que, al estar aplicados fundamentalmente a Sistemas Orientados a Objetos, el papel de meta-entidad lo jugarán objetos – que representan al referente exponiendo su comportamiento base, por ejemplo, la forma en que gestiona sus mensajes.

De acuerdo con esta aproximación a la reflexión, la semántica del paso de mensaje puede definirse enviando un mensaje al meta-objeto.

Esta aproximación plantea nuevamente las eternas cuestiones ya expuestas en el apartado **Definición de una Arquitectura Reflectiva**, ¿Cuál es la naturaleza del meta-objeto? y ¿Cuándo se pasa el control del nivel base al meta-nivel?.

---

[1] En lo que sigue, nos referiremos siempre a la reflectividad del comportamiento a menos que, explícitamente, se diga lo contrario.

Dependiendo de cómo se responda a tales preguntas es posible distinguir 2 aproximaciones.

- **Modelo MetaClase**. Se basa en la equivalencia entre meta-objeto y la clase de un objeto.
- **Modelo MetaObjetos**. En el que los meta-objetos son instancias de una clase determinada, normalmente denominada **MetaClase**.

Exposición de la Comunicación

El segundo modelo se basa en la exposición de la comunicación en sí, de forma que cada mensaje se convierte en un objeto y, por tanto, reacciona al mensaje **SEND**.

En lo que sigue se analizarán el Modelo MetaClase y el Modelo MetaObjeto como pertenecientes a la primera corriente y la exposición de mensajes o Modelo MetaComunicación a la segunda.

5.1 Modelo MetaClase

Según la definición de [Gra89], *una clase describe tanto la estructura como el comportamiento de sus instancias. La estructura es el conjunto de variables cuyo valor estará asociado a las instancias. El comportamiento es el conjunto de métodos a los que las instancias de la clase serán capaces de responder*.

El modelo de **MetaClase**[Coi87, BC89] se basa en la equivalencia entre la clase de un objeto y su meta-objeto y, por tanto, implementa la torre reflectiva siguiendo la cadena de instanciaciones.

Las clases serán objetos, instancias de una clase, su **MetaClase**. A su vez, las MetaClases forman una jerarquía que deriva de la clase raíz, **CLASS**.

De esto se deriva que una clase es un meta-objeto que expone una entidad base. De hecho, en los sistemas basados en el paradigma de clase/instancia, la clase de un objeto es considerada normalmente como su meta-objeto, debido a la reflectividad estructural del modelo.

De esta forma, para un objeto O es equivalente, desde el punto de vista del comportamiento, recibir un mensaje M o que su clase reciba el mensaje indicado para gestionar un mensaje, que, en la mayoría de las obras se representa como **HANDLEMSG**.

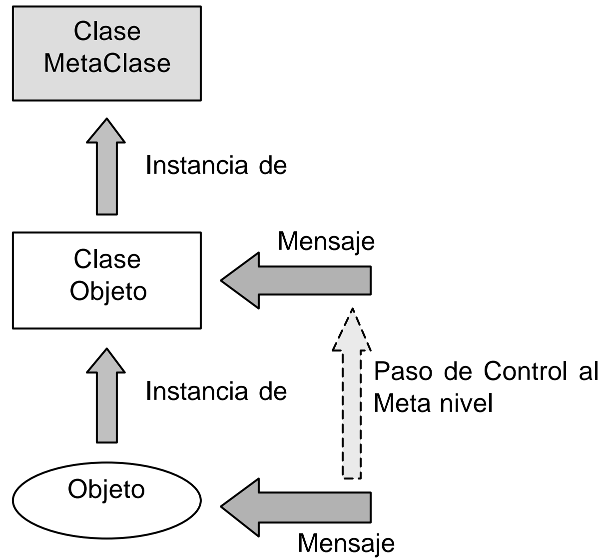


Figura 12.4 Esquema del modelo Meta-Clase

El método HANDLEMSG se define en la meta-clase y el método de gestión de mensajes por defecto se describe en la meta-clase raíz de la jerarquía de metaclases –**CLASS**-, de la que son instancias las clases.

Cuando un objeto recibe un mensaje, debe decidir si utilizar el mecanismo por defecto o pasar el control al meta-nivel. Esto puede lograrse mediante una entrada definida en el receptor, lo que permite lograr la reflectividad para un objeto, o en su clase, lo que establece un modo reflectivo para todas las instancias.

5.1.1 Ventajas

Sencillez

La clase encaja perfectamente con el rol de controlador y modificador de la información estructural porque mantiene esa información. La implantación de este modelo es posible, únicamente en aquellos lenguajes que manejen las clases como objetos como son Smalltalk[FJ89] o CLOS[BGW03].

5.1.2 Inconvenientes

Se han señalado algunos inconvenientes a este modelo[Fer89].

Especialización

El problema radica en la especialización del meta-comportamiento de una instancia concreta. Todas las instancias de una clase tienen el mismo meta-objeto, por tanto, todas tienen el mismo meta-comportamiento.

Para especializar el meta-comportamiento para una instancia, es posible usar la herencia construyendo una clase nueva que difiera de la clase original sólo en el meta-comportamiento, o bien mantener diccionarios que mantengan información y métodos de cada instancia.

Algunos intentos de añadir reflectividad a la máquina virtual Java [GK97a] tienen este problema.

Cambio dinámico

Otro problema radica en la dificultad de cambiar dinámicamente el comportamiento de un objeto. Tal cambio implicaría sustituir su meta-clase por otra.

Sin embargo, no todos los lenguajes permiten el cambio dinámico de la clase de un objeto. Es más, el cambio de la clase de un objeto puede derivar en inconsistencias.

Parcialmente meta

Así mismo, tampoco es posible registrar características particulares de cada objeto en su meta-representación. Esta limitación es crucial dado que uno de los aspectos más importantes de las meta-entidades es su habilidad para almacenar información acerca de su objeto base o referente.

Compatibilidad

Los lenguajes basados en clases organizan las aplicaciones en jerarquías de clases utilizando para ello la herencia, lo que además, facilita la reutilización [Weg90].

Aquellos lenguajes que utilizan el concepto de clase para la reflectividad tienen que tener en cuenta, además del enlace clase-subclase-superclase, el enlace de instancia existente entre una clase y su MetaClase.

La interacción entre la herencia y la instanciación debe tenerse muy en cuenta a la hora de diseñar librerías de clases. El problema radica en determinar si una clase puede heredar de otra o no, partiendo del hecho de que ambas clases son instancias de MetaClases diferentes.

Esto da lugar a la aparición del problema de la compatibilidad de meta-clases como se apunta en [Gra89, BLR96].

- **Compatibilidad Ascendente**. Considérese la situación representada en la Figura 12.5(a), donde una clase A, instancia de una meta-clase MetaA, implementa un método m que envía el mensaje m' a la clase de la instancia que recibe el método m.

  Supóngase una clase B, derivada de A e instancia de una metaclase MetaB. Si entre MetaA y MetaB no existe ninguna relación, es factible suponer que el mensaje m' sólo sea conocido por la primera de las dos metaclases. Cabe entonces preguntarse qué sucederá cuando una instancia de B reciba el mensaje m, lo que provocará que MetaB reciba el mensaje m'.

- **Compatibilidad Descendente**. Considérese la situación representada en la Figura 12.5(b), donde una clase A, instancia de una meta-clase MetaA, implementa un método m y MetaA implementa el método m' que crea una nueva instancia del receptor y le envía el mensaje m. ¿Qué sucederá cuando se envíe el mensaje m' a la clase B que es instancia de la metaclase MetaB, que hereda de MetaA?

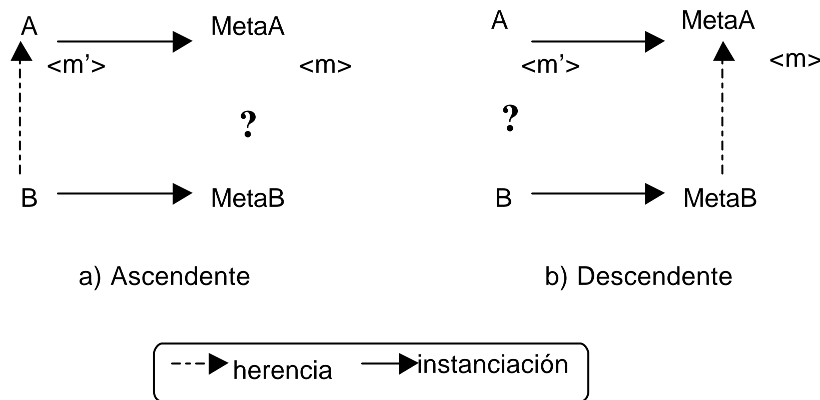


Figura 12.5 Problemas de Compatibilidad del modelo Meta-Clase

5.2 Modelo MetaObjeto

En el modelo de **MetaObjetos** [KRB91], los meta-objetos son instancias de una clase especial, la clase **MetaObjeto** o de una de sus subclases. Los meta-objetos no se identifican con la clase del objeto y únicamente incluyen información de control del comportamiento, dejando la información estructural a la clase. De esta forma, la torre reflectiva se lleva a cabo siguiendo la relación cliente/servidor.

De esta forma cuando un objeto O recibe un mensaje, es equivalente a enviar el mensaje **HANDLEMSG** a su meta-objeto.

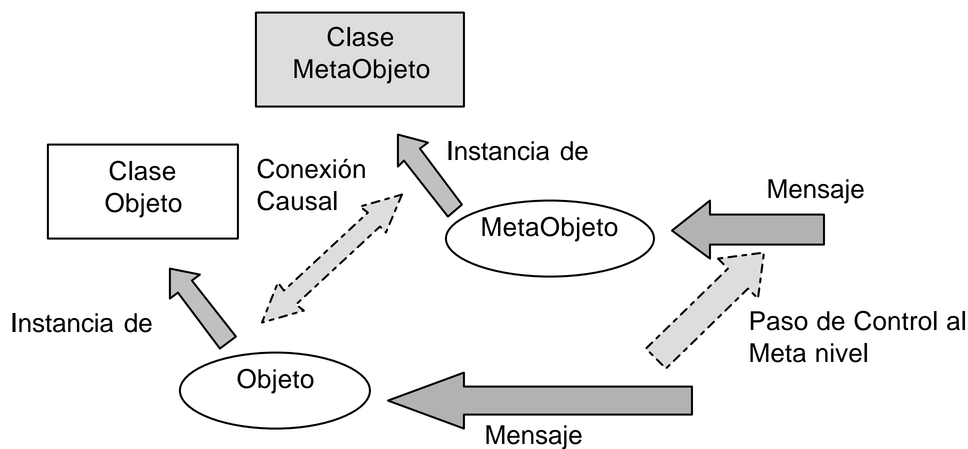


Figura 12.6 Esquema del Modelo Meta-Objeto

Entidades independientes gestionan tanto la introspección como la intercesión de cada entidad base. Cada meta-objeto intercepta los mensajes enviados a su objeto y ejecuta su meta-computación antes de reenviarlos a su objeto base.

En este modelo no existen, en principio, restricciones ni imposiciones sobre la relación entre las entidades base y las meta-entidades. Cada meta-objeto puede estar conectado a varios objetos del nivel base y cada objeto puede tener enlazados varios meta-objetos (uno a la vez) durante su ciclo de vida.

El Meta-Espacio y el Reflector

Es posible también que cada objeto tenga asociados varios meta-objetos cada uno de los cuales se refiere a una parte concreta de su comportamiento. Al conjunto de meta-objetos que componen el entorno de ejecución de un objeto se le denomina **meta-espacio** y suele haber un punto de entrada al meta-espacio denominado **reflector** [Yok92].

El meta-espacio puede verse como un **micronúcleo personalizable** dado que, reemplazando los meta-objetos que lo componen es posible adaptar (personalizar) la implementación del sistema para la aplicación concreta.

Soporte en el entorno

Con el fin de tratar con meta-objetos, es necesario efectuar algunos cambios en el entorno. El primero consiste en modificar la representación en tiempo de ejecución del objeto, la **clase Objeto**, para enlazar con su meta-objeto. De esta forma es posible cambiar el meta-objeto simplemente cambiando el valor asociado a esa posición del objeto.

El segundo cambio se refiere a la definición de la función primitiva de paso de mensajes, que debe chequear si hay un meta-objeto asociado al objeto, en cuyo caso, el mensaje se delega al meta-objeto.

5.2.1 Ventajas

Elimina, prácticamente todos los inconvenientes del modelo anterior.

Especialización del meta-comportamiento

Es sencillo especializar el meta-comportamiento de un objeto individual desarrollando una nueva clase de meta-objetos que refinen el comportamiento original de alguna forma.

Monitorización del objeto

También es sencillo que un meta-objeto monitorice el comportamiento de su objeto, registrando qué sucede y, eventualmente, decidiendo algún cambio.

Modificación del meta-objeto

Partiendo de que los meta-objetos son objetos como cualquier otro, que se crean y destruyen de la misma forma que otros objetos, con un identificador único y conocido y sabiendo que se enlazan con su objeto base a través de alguna asociación con este identificador único, es posible modificar fácilmente el meta-objeto de un objeto asignando a tal asociación el identificador de otro meta-objeto con un comportamiento diferente.

5.2.2 Inconvenientes

El mayor inconveniente reseñable de este modelo es que un meta-objeto puede monitorizar un mensaje sólo cuando ya ha sido recibido por su objeto referente. Por tanto pierde información acerca del emisor.

Este es el modelo más usado no sólo en lenguajes de programación (3-KRS) sino también es sistemas operativos ApertOS[Yok92], sistemas distribuidos como CodA[McA95a], etc.

6 La Reflectividad como Exposición de la Comunicación o Exposición de los Mensajes

Esta última aproximación, consiste en la exposición de la comunicación en sí. Cada comunicación puede ser un objeto y, por tanto, atiende o reacciona al mensaje **SEND**.

Los Objetos Mensaje

En este modelo, las meta-entidades son objetos especiales, denominados **mensajes**, que exponen las acciones que deberían ejecutar las entidades base[Fer89].

La clase **MESSAGE** contiene toda la información necesaria para interpretar el mensaje. Cuando se invoca un método, se crea una instancia de la clase **MESSAGE** y se envía el mensaje **SEND** a tal instancia. El objeto mensaje está dotado con su propia gestión de acuerdo al tipo de meta-computación requerida. Cuando la meta-computación termina, se destruye el objeto.

La clase de un mensaje define el meta-comportamiento realizado por el mensaje de forma que diferentes mensajes pueden tener diferentes clases. Es posible definir diferentes comportamientos para llamadas a métodos para cada objeto, especificando un modo diferente para cada llamada a método.

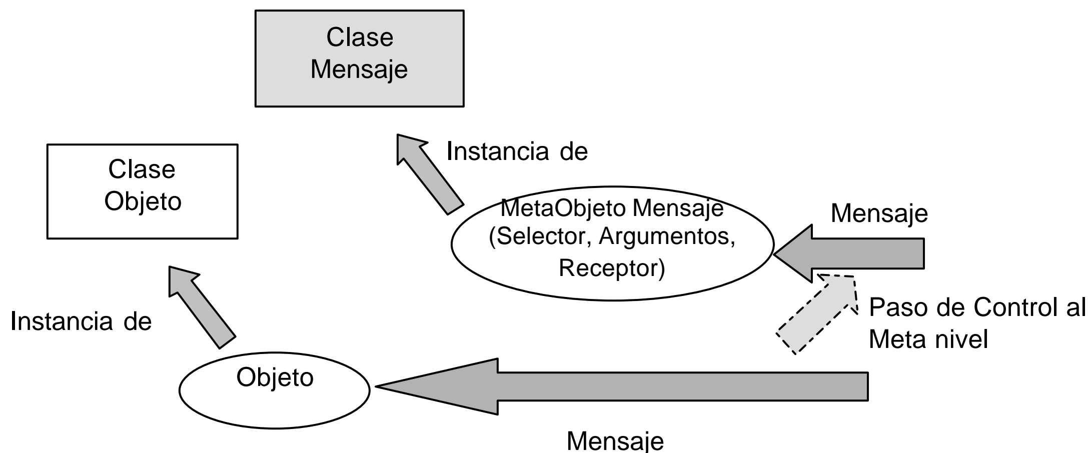


Figura 12.7 Esquema del Modelo de Exposición de Mensajes

La Torre Reflectiva

Este modelo tiene una torre reflectiva compuesta únicamente por dos niveles, el nivel base y el meta-nivel, que expone los mensajes.

Transferencia de Control

En esta aproximación el sistema pasa el control al meta-nivel para todos los envíos de mensajes excepto cuando recibe el mensaje **SEND** para evitar la regresión infinita.

De este modo, el sistema está casi siempre en modo reflectivo.

6.1.1 Ventajas

La primera ventaja destacable es la facilidad que ofrece para diferenciar entre distintos tipos de mensajes: se pueden definir y usar distintas subclases de comunicación en lugar de la clase **MESSAGE** estándar.

Por otro lado, el uso de diferentes tipos de mensajes permite una extensión incremental del sistema. Es posible incorporar fácilmente nociones como paso concurrente de mensajes, etc.

6.1.2 Inconvenientes

La principal desventaja de este modelo es que los meta-objetos mensaje no están enlazados de ninguna manera con las entidades base que los originan y, por tanto, no pueden acceder a su información estructural.

Además, los meta-objetos mensaje tienen una vida muy corta, existen sólo durante el tiempo que dura la acción que representa.

Debido a todo esto, son inadecuados para monitorizar el comportamiento de los objetos del nivel base o almacenar información sobre la meta-computación (**carencia de continuidad**), para representar información específica acerca de los objetos o representar el comportamiento de un objeto determinado.

Sin embargo, es una aproximación hábil para combinarse con los modelos anteriores.

7 Meta-Interfaces y MOP

El modelo de la **implantación abierta** propuesto por Kiczales[Kic92] argumenta que, mientras que el modelo tradicional de ocultación de información protege a los clientes de una abstracción de tener que conocer los detalles de cómo está implementada esta, también evita que pueda alterar algunos detalles cuando lo necesite.

Así pues, se manifiesta como conveniente, el que los clientes de una abstracción puedan tener algún control sobre decisiones tomadas en esa implantación.

Sin embargo, esta posibilidad de adaptación del entorno de ejecución no debe suponer quebraderos de cabeza para los usuarios. En este punto, no está de más recordar que la razón original de intentar esconder los detalles de implantación era una muy buena razón: simplificar la tarea del programador abstrayendo aquellos detalles con los que se suponía que no tendría que trabajar[KL93].

7.1 Interfaz base versus Interfaz Meta

Para permitir la adaptación sin incrementar la complejidad, se propone separar la interfaz que presenta un servicio o abstracción en **interfaz base** o **protocolo base**, que ofrece a los usuarios la funcionalidad usual de la abstracción o servicio, y **meta-interfaz** o **meta-protocolo**, que permite a los clientes controlar cómo se implementan ciertos aspectos de la abstracción o servicio [KL93, GC95, HPM+97].

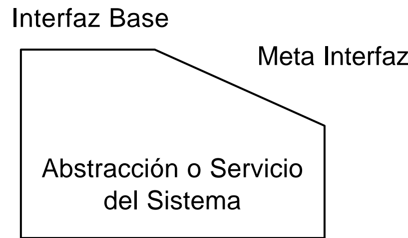


Figura 12.8 Interfaz Base versus Meta Interfaz

Por tanto, en lugar de exponer todos los detalles de la implantación, la meta interfaz debería proporcionar una visión abstracta de la misma que permitiese a los clientes ajustarla en modos bien definidos.

El objetivo final de separar el comportamiento y la implantación es permitir al programador cliente:

- ser capaz de controlar la implantación del servicio cuando lo necesite
- ser capaz de tratar el comportamiento y la implantación por separado.

Ofreciendo interfaz base y meta-interfaz por separado, se ofrece al usuario programador una visión separada de ambos mundos.

7.1.1 Interfaz base

La interfaz de la aplicación se modela como una caja negra tradicional que expone la funcionalidad y no los detalles de implantación.

Por ejemplo, en el caso de la memoria virtual, la interfaz base únicamente lee y escribe bytes, en el caso de un sistema de ficheros abre/cierra ficheros, lee/escribe bytes, etc. Los clientes del servicio escriben los programas sobre la interfaz base de modo tradicional.

7.1.2 Meta-interfaz

La meta-interfaz articula y expone los detalles de implantación posibilitando a los clientes controlar las decisiones a tomar a la hora de establecer la correspondencia entre un servicio y su implantación final, permitiendo así conseguir una **personalización de grano fino**.

Por ejemplo, en el caso de una meta-interfaz al sistema de memoria virtual, el cliente escribiría código utilizando la interfaz base y, en caso de que el sistema de paginación por defecto ofreciese un rendimiento pobre para el comportamiento del cliente, este podría escribir código usando la meta-interfaz para indicar un algoritmo alternativo para el comportamiento del código base.

7.2 Meta-Object Protocols (MOPS)

En esencia, un **MOP** [KRB92, GC96], especifica la implantación de un modelo de objetos reflectivo.

Si se considera que el meta-nivel es una implantación del modelo de objetos soportado por el lenguaje, un MOP especifica **qué objetos del meta-nivel son necesarios para implantar el modelo de objetos. Y la interfaz exportada por cada uno de esos objetos forma parte de la interfaz del MOP**[GC95, GC96].

En un sistema orientado a objetos donde, tanto las aplicaciones como el propio sistema, se modelan y representan en tiempo de ejecución como objetos que interaccionan entre sí, el meta-nivel especifica la implantación del modelo de objetos soportado por el sistema en términos de los meta-objetos necesarios para lograr el comportamiento deseado del sistema [KRB97].

El Meta-Protocolo o meta-interfaz se refiere, no a la especificación de la interfaz del objeto, sino a la especificación de la implantación de los objetos.

Entre los ejemplos más destacados cabe citar el MOP de CLOS[KRB91], o el MOP de CodA[McA93].

8 Reflectividad y Sistemas Operativos

Una aproximación novedosa en la arquitectura de un sistema operativo son las arquitecturas orientadas a objeto, donde cada componente del sistema operativo se define y encapsula como un objeto y, por tanto, el mecanismo de invocación entre objetos es el único mecanismo de interacción en esta arquitectura [TNO92].

Arquitectura Reflectiva de un Sistema Operativo basado en Meta-Objetos

La aplicación de la reflectividad a la definición de la arquitectura del sistema operativo mediante la extensión de la aproximación orientada a objeto basada en Meta-Objetos [TNO92, KL93], supone que, si bien la funcionalidad se ofrecerá en forma de objetos o meta-objetos, existirá una separación clara entre unos y otros.

Mientras los primeros definen únicamente tareas de la aplicación a nivel usuario, el conjunto de meta-objertos define el comportamiento de los objetos de usuario, su entorno de ejecución. De esta forma, una llamada al sistema tradicional se transforma en una invocación a un método ofrecido por un meta-objeto [LYI95].

Por ejemplo, en la arquitectura reflectiva definida en Apertos[Yok92], un conjunto de meta-objetos define el comportamiento de cada objeto de usuario y este último, puede moverse entre dos conjuntos de meta-objetos que ofrezcan diferentes servicios.

Separación Objeto/Meta-Objeto

La computación de los objetos de usuario se gestionará mediante un conjunto de objetos, los meta-objetos, que definen el modelo de objetos de los primeros[TNO92].

Por ejemplo, en Apertos[Yok92] un grupo de meta-objetos forman un meta espacio y el punto de entrada al meta espacio se denomina reflector. Un objeto de usuario puede migrar entre diferentes meta-espacios manteniendo siempre la compatibilidad. Por ejemplo, si un objeto de usuario necesita moverse a almacenamiento secundario, el objeto debería migrar a un meta-espacio que dé soporte a la persistencia.

La diferencia entre el núcleo tradicional y la arquitectura reflectiva es que el enlace entre objetos y meta-objetos es dinámico de tal forma que la aplicación puede cambiar el comportamieto de los objetos cambiando su meta-espacio.

9 Reflectividad en sistemas concurrentes orientados a objetos

Una metodología de diseño ampliamente estudiada denominada **Object-Based/Oriented Concurrent Programming** [YT87b], propugna el diseño del sistema como una colección de objetos que se ejecutan concurrentemente y que interactuan unos con otros mediante el envío de mensajes. Estos objetos reciben el nombre de **objetos concurrentes**.

Sin embargo, en esta metodología de diseño no se estudia ni propone ningún medio para permitir la adaptación del sistema subyacente adecuándolo a los requerimientos particulares que pueda presentar una aplicación. La intención última de este diseño es conseguir que el sistema explote el mayor grado de paralelismo posible.

Un sistema concurrente incluye gran cantidad de aspectos que no se presentan en los sistemas secuenciales como planificación, comunicación, balance de carga, etc. En aquellos sistemas basados en una arquitectura tradicional estos aspectos se implementan de forma fija y estática – *ad-hoc* – en el sistema y el control que se puede establecer sobre ellos es fijo y con pocas posibilidades para la extensibilidad.

Cuando se ejecuta una aplicación en un sistema, las mejores decisiones de control se toman mediante la interacción entre la arquitectura subyacente o sistema subyacente y la aplicación.

Un ejemplo son las **Scheduler Activations** [ABL+91] en las que las decisiones de planificación se dividen entre el sistema operativo y la aplicación que se está ejecutando.

Sin embargo, las arquitecturas tradicionales permiten a la aplicación un conocimiento y una influencia muy limitadas en el comportamiento del sistema subyacente. Por ejemplo, algunas llamadas al sistema en sistemas tradicionales permiten cierta influencia pero limitada, inflexible y de una forma no modular. Así por ejemplo, SunOS con *madvise* permite a los clientes elegir entre un conjunto de implantaciones predefinidas, en lugar de permitirle escribir su propia implantación.

En un sistema concurrente, esto es mucho más grave que en un sistema secuencial ya que los sistemas concurrentes son mucho más complejos y deben estar abiertos a extensiones y modificaciones para adaptarlo a nuevos entornos.

Con una arquitectura reflectiva, la aplicación puede influir en el sistema de forma modular y uniforme[YW89, MWI+91]. El modelo de una arquitectura reflectiva se presenta como una colección de objetos o meta-objetos y la utilización de las capacidades reflectivas se lleva a cabo enviando un mensaje al meta-objeto correspondiente.

Visión General del Modelo de Computación

En un **modelo de computación concurrente orientado a objetos**, el sistema es una colección de objetos, que se comportan como entidades autónomas de procesamiento de información[WY88].

Cada objeto, en el caso más habitual, está dotado de capacidad de computación serie que se activa mediante el envío de un mensaje o invocación de un método. Básicamente, las secuencias de acciones en los distintos objetos proceden de forma asíncrona, o lo que es lo mismo, muchos objetos pueden estar ejecutando algunos de sus métodos en paralelo. Los objetos se convierten así en unidades de concurrencia.

Desde el punto de vista de programación del modelo, un objeto es un bloque básico de construcción de programas, que es básicamente, la descripción de las acciones a realizar por el objeto cuando reciba un mensaje.

Estructura del Objeto

Dado que cada objeto tiene una capacidad de procesamiento serie, ejecuta una acción al tiempo. Debido a esto, aquellos mensajes que se reciban cuando el objeto esté procesando uno anterior, deben ser pospuestos. Por tanto, cada objeto está dotado de un estado y un conjunto de métodos, además de una cola de mensajes y un evaluador de los mismos.

Envío, Recepción y Procesamiento de los Mensajes

Toda transmisión de mensajes es asíncrona, sin necesidad de acuerdo (del inglés *handshaking*) para enviar/recibir mensajes. Esto significa que un objeto puede enviar un mensaje cuando quiera, independientemente del estado del objeto receptor.

Cuando se envía un mensaje a un objeto, este lo tratará de acuerdo a lo siguiente:

- **Llegada**. El mensaje llega al objeto receptor que comienza el procesamiento del mensaje. Se supone que una vez se transmite el mensaje, la llegada del mismo está garantizada por el sistema de distribución[ATD+98a, ATD+98b].
- **Recepción**. El receptor comienza el procesamiento del mensaje, si es posible, o lo encola en la cola de mensajes.
- **Aceptación**. El receptor busca un método apropiado para dar servicio al mensaje solicitado. Si se acepta el mensaje, se comienza la ejecución del método.
- **Fin procesamiento**. Cuando termina la evaluación o procesamiento del mensaje, el objeto receptor selecciona otro mensaje.

Reflectividad en el modelo

La representación de un objeto se construye a partir de la representación de un sistema de computación serie. Esta representación contiene una **parte estructural**, la cola de mensajes, el conjunto de métodos o el estado, y una **parte computacional**, la llegada, recepción, aceptación y procesamiento de los mensajes.

La reflectividad en un Sistema Concurrente Orientado a Objeto puede introducirse mediante meta-objetos que representan ambos aspectos de los objetos del nivel base.

La computación reflectiva de un objeto la lleva a cabo su meta-objeto y, para que esto tenga lugar, el objeto base envía mensajes a su meta-objeto. Esto permite al objeto base inquirir sobre su estado interno y modificarse a sí mismo, a través de su meta-objeto, dado que los aspectos estructurales y de comportamiento del objeto base están representados en su meta-objeto.

Además, un meta-objeto puede recibir mensajes de otros objetos que no sean su referente. En un sistema que consiste en una colección de objetos, es posible considerar

el meta-objeto de un objeto como la representación parcial de tal sistema. Por tanto, la computación reflectiva en el sistema se lleva a cabo enviando mensajes a los meta-objetos.

10 Ventajas de una arquitectura reflectiva aplicada a un sistema concurrente Orientado a Objetos

Todos los trabajos que versan sobre la aplicación de la reflectividad en distintos sistemas[YW89, Smi90, KG96, CS98], ponen mucho énfasis en las ventajas que ello ofrece.

Existen una serie de ventajas fundamentales derivadas de construir un sistema concurrente con una arquitectura reflectiva que se exponen a continuación.

Mantenimiento de la uniformidad conceptual

La representación del entorno de ejecución de los objetos se realiza por medio de un conjunto de entidades, objetos a su vez, como por ejemplo el sistema CodA[McA95a].

De esta forma, las abstracciones y mecanismos que proporcionan la funcionalidad del sistema operativo se proporcionan en forma de objetos (única abstracción en el sistema).

Extensibilidad y adaptabilidad

La utilización de la reflectividad establece una **arquitectura abierta**. Un sistema con una arquitectura reflectiva proporciona un medio explícito y modular para modificar dinámicamente la organización interna del sistema.

Esto permite al programador cierto control sobre aspectos computacionales del sistema de forma modular y dinámica, permitiendo con ello implementar nuevas políticas sin cambiar el código de la aplicación.

Dado que varias tareas de control y gestión de recursos se modelan mediante meta-objetos, se pone a disposición de las aplicaciones la posibilidad de gestionar de forma modular cuestiones como la política de planificación, la migración de objetos, y otras.

Separación de asuntos (concerns)

Uno de los principios de diseño más importantes a la hora de diseñar sistemas operativos consiste en separar la política de los mecanismos[Cro96]. En este caso, la utilización de la reflectividad como medio de estructurar el sistema promueve la aplicación de este principio[KG96].

En la programación tradicional, los programas se mezclan y se ven complicados por el código que representa los algoritmos que los gestionan. Esto dificulta considerablemente entender, mantener, depurar y validar los programas.

Una arquitectura reflectiva proporciona interfaces a cada uno de los niveles de funcionalidad: una interfaz al nivel base y una interfaz al metanivel.

Esta separación del nivel base (algoritmo base) y el meta-nivel (nivel reflectivo) facilita la reutilización de políticas sin dañar el código base. El uso de un meta-espacio separado permite al programador concentrarse en el dominio de la aplicación. La

funcionalidad adicional se suministra mediante meta-componentes, que trabajarán con los objetos definidos por la aplicación.

Favorece el diseño de un modelo de objetos activo

La implantación de un modelo de objetos activo se consigue fácilmente extendiendo la semántica tradicional del objeto, ampliamente extendida y conocida de los modelos de objetos pasivos, que definen los objetos como meros contenedores de datos y dotando a estos de un contenido semántico mucho mayor, que favorecerá el incremento del paralelismo.

Así, cada objeto deberá estar compuesto de datos + métodos + computación. Mientras que la parte pasiva del objeto implementa los aspectos funcionales del mismo, el comportamiento activo del objeto se define en el meta-nivel [GK97b].

De la misma forma, el control de la concurrencia puede hacerse de forma extensible, permitiendo que un mismo objeto base pueda implementarse con distintas políticas de control en el meta-nivel.

Incremento de la productividad

Igual que la programación OO, la meta-programación es un nuevo paradigma que empuja hacia programas más estructurados (separación del código funcional del no funcional) y mantenibles.

La combinación de ambas, facilita la construcción de programas más estructurados, de más fácil mantenimiento, más adaptables y robustos.

Configurabilidad

El soporte de un meta-nivel facilita una arquitectura abierta[Kic96] en la que se pueden implementar nuevas políticas sin cambiar el código de la aplicación.

Esta propiedad se consigue, no sólo a nivel de los desarrolladores de la aplicación, que se benefician de la meta-programación para desarrollar aplicaciones configurables, sino también a nivel de los usuarios, que pueden reemplazar meta-componentes para personalizar las aplicaciones de acuerdo a sus necesidades específicas.

Generalización para multiprocesadores y sistemas distribuidos

La separación del código de aplicación del nivel base del comportamiento del objeto representado en el meta-nivel, juega un papel fundamental en la distribución y permite a los desarrolladores de aplicaciones distribuidas prototipar sus aplicaciones y experimentar con distintos modelos de distribución, minimizando los efectos en el código de la aplicación [McA95b].

Por tanto, los objetos normales son fácilmente reutilizados en entornos distribuidos, gracias a que la reflectividad permite la adición de comportamiento distribuido en el meta-nivel, de forma transparente.

Además, es relativamente sencillo integrar nuevos mecanismos de distribución.

Capítulo XIII

Sistemas Reflectivos

No es posible escribir un capítulo sobre reflectividad sin mencionar algunos de los sistemas más nombrados al respecto como pueden ser Smalltalk. También se introduce en este breve compendio sobre los sistemas reflectivos algunos trabajos sobre Java, por el impacto que está teniendo en el campo de las máquinas abstractas y, por último, un breve comentario sobre SOM, por ser un claro ejemplo de la utilización de metaclases para la implantación de la reflectividad.

1 Reflectividad Estructural. La API Reflectiva de Java

La versión 1.1 de Java [Sun97b] proporciona una nueva API, denominada **API Reflectiva**, del inglés *Reflection API*, que proporciona la facilidad de introspección sobre las clases y objetos de la Máquina Virtual Java (JVM).

La API reflectiva de Java representa, o refleja, clases, interfaces y objetos de la implantación de la JVM lo que permite determinar la clase de un objeto, conseguir información acerca de los métodos, constructores, superclases, etc. de una clase, e incluso invocar un método de un objeto aunque dicho método no se conozca hasta el momento de la ejecución.

Si la política de seguridad lo permite, esta API ofrece un conjunto de operaciones restringido que permiten acceder y modificar en cierto modo el entorno en tiempo de ejecución. Las posibles actuaciones sobre el entorno incluyen:

- Construir nuevas instancias de clases y nuevos vectores y matrices
- Acceder y modificar campos de objetos y clases
- Invocar métodos en objetos y clases
- Acceder y modificar elementos de los vectores.

1.1 Modelo Reflectivo

Fundamentalmente, esta API consta de dos componentes. Por una lado, los **objetos que representan las distintas partes de un fichero de clase** y por otro, **un medio para extraer tales objetos de forma segura**[McMa97].

Para representar información en tiempo de ejecución, la API reflectiva de Java define nuevas clases y métodos.

1.1.1 Nuevas Clases: Objetos que representan el entorno en tiempo de ejecución

La API Reflectiva de Java define nuevas clases para representar los distintos componentes que forman la representación en tiempo de ejecución de un objeto.

Los objetos instancias de estas clases se usan para manipular los objetos físicos subyacentes, inspeccionando y modificando valores, invocando métodos en objetos y clases y creando nuevas instancias de clases.

Field, **Method** y **Constructor**, reflejan los miembros y constructores de las clases e interfaces. Estas clases proporcionan información reflectiva acerca del elemento subyacente y un medio para usarlo y actuar así sobre los objetos Java.

- **Field** refleja un campo que puede ser una variable de clase o una variable instancia. Los métodos de esta clase permiten obtener el tipo del campo subyacente y conseguir y modificar los valores de los campos en los objetos.
- El objeto **Method** refleja los métodos de una clase. Los métodos de la clase Method se pueden utilizar para obtener los tipos de los parámetros, el tipo de retorno y los tipos de excepciones lanzadas por el método reflejado. Introduce un método invoke que permite invocar el método subyacente en los objetos usando resolución dinámica basada en la clase en tiempo de ejecución del objeto y en la clase del método reflejado, su nombre y parámetros.
- El objeto **Constructor** refleja el constructor de una clase. Sus métodos permiten obtener los tipos de los parámetros y las excepciones del constructor subyacente. El método NewInstance de la clase Constructor puede utilizarse para crear e inicializar una nueva instancia de la clase que declara el constructor, en tiempo de ejecución.

Array, proporciona métodos para construir, inspeccionar, acceder y manipular dinámicamente a arrays Java. Y, por último, **modifier**, proporciona métodos para decodificar modificadores, como static o public, devueltos por las clases y sus componentes, que están codificados como enteros.

1.1.2 Nuevos Métodos: Conseguir la información

Además de crear nuevas clases para representar la información en tiempo de ejecución del objeto, también define nuevos métodos para inspeccionar y manipular esa información.

El mecanismo para acceder a dicha información se construye en la clase **Class**. Esta es el tipo universal para la meta-información, y describe los objetos dentro del sistema Java.

Para conseguir información en tiempo de ejecución, basta añadir nuevos métodos en la clase **Class** (getDeclaredConstructor, getDeclaredConstructors, getMethod, getMethods, getField y getFields) que proporcionan el medio para construir nuevas instancias de las clases Field, Method y Constructor.

Se añaden métodos para determinar si un objeto representa un array o un tipo primitivo, métodos que devuelven la representación de los modificadores del tipo de la clase, métodos que devuelven objetos representando los campos, métodos y constructores y métodos que determinan si una clase o interfaz es superclase de otra dada o si un objeto es instancia de un tipo.

1.2 Componentes

Un **JavaBean** se puede definir [Fla97], como un modelo de componentes software reusables que puede ser manipulado.

Un JavaBean posee la habilidad de publicar sus métodos, operaciones y propiedades, es decir, tiene la propiedad de introspección, mediante las clases e interfaces ofrecidas por el paquete java.beans que permite el acceso a los métodos, la cabecera de los mismos y otras propiedades de forma segura.

La introspección también permite a un Bean descubrir los métodos, operaciones y propiedades de otros beans.

La introspección en los JavaBeans se basa en la utilización de la Java Reflection API, es decir, la construcción del paquete java.beans se basa en la utilización de la API descrita anteriormente.

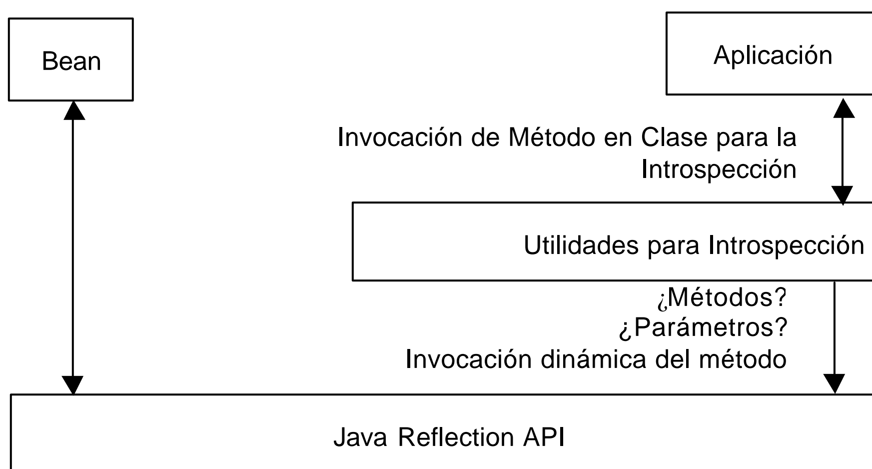


Figura 13.1 API Reflectiva de Java: Introspección y Componentes

2 MetaJava

MetaJava [Gol97] es un sistema desarrollado sobre la JVM[Sun97a], que extiende el intérprete de Java permitiendo, no sólo reflectividad estructural, sino también reflectividad del comportamiento.

2.1 Guías de Diseño

El principio de diseño fundamental en MetaJava es que **el meta-sistema no debe efectuar acciones triviales, sino excepcionales**.

De ello se derivan algunos criterios de diseño:

Rendimiento. Si no se utiliza, el meta sistema no debe imponer ninguna penalización de rendimiento.

- Permitir en lo posible, la separación de los aspectos funcionales, específicos de la aplicación, de los no funcionales, como persistencia o replicación. Esta separación debe ser visible desde la fase de diseño a la de codificación.

- Tanto los programas del nivel base como los programas del meta nivel debe ser reusables.
- La arquitectura debe ser general, es decir, problemas como distribución, sincronización y otros deben tener cabida y solución en ella.

2.2 Modelo Computacional de MetaJava

Los sistemas tradicionales consisten en un sistema operativo y, sobre él, un programa que explota los servicios del mismo usando una API.

La aproximación reflectiva propuesta por MetaJava es diferente. El sistema consiste en el Sistema Operativo, el programa de aplicación (**sistema base**) y el **meta sistema**.

2.2.1 Separación en Nivel Base y Meta Nivel

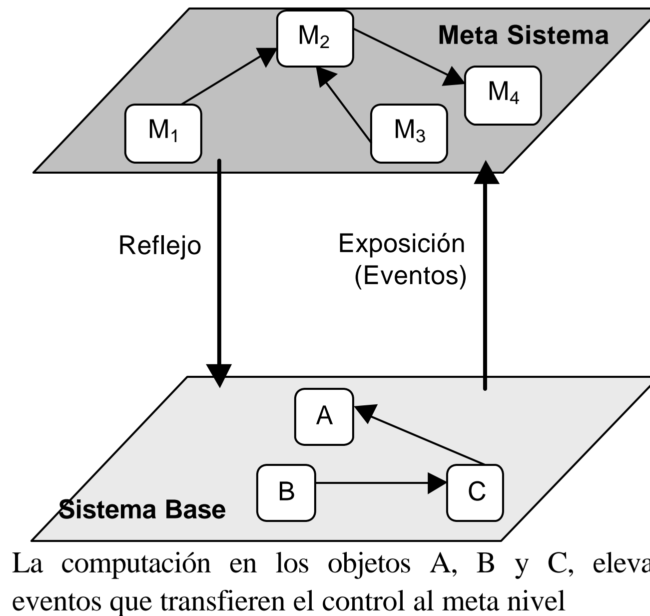
El programa está dividido en sistema base y meta sistema, cada uno con un ámbito claramente separado,

- **Sistema Base**. El programa de aplicación, o sistema base, no debe preocuparse del meta sistema, sino dedicarse al problema de la aplicación.
- **Meta Sistema**. El meta sistema se encarga de las computaciones que no solucionan directamente el problema de la aplicación, sino que solucionan los problemas de la computación en el nivel base.

2.2.2 Transferencia de Control

Reflectividad Implícita

Para comunicar el nivel base y el meta nivel MetaJava utiliza la emisión de eventos síncronos, es decir, la computación en el nivel base eleva eventos que se envían al meta sistema.



La computación en los objetos A, B y C, eleva eventos que transfieren el control al meta nivel

Figura 13.2 Modelo Computacional de Reflectividad del comportamiento

El Meta sistema evalúa los eventos y reacciona de manera específica a ellos. Todos los eventos se gestionan de forma síncrona y la computación del nivel base se suspende mientras el meta objeto procesa el evento. Esto da al meta nivel control absoluto sobre la actividad del nivel base.

Por ejemplo, si el meta nivel recibe el evento enter-method, el comportamiento por defecto sería ejecutar el método. Sin embargo, el meta-objeto podría también sincronizar la ejecución del método con otro método del objeto base. Otras alternativas serían encolar el método para retrasar su ejecución y retornar al llamador inmediatamente o ejecutar el método en otra máquina. Lo que suceda depende, completamente, del meta objeto utilizado.

Reflectividad Explícita

Un objeto base también puede invocar directamente un método en el meta objeto. Es lo que se conoce como **meta interacción explícita** y se utiliza para controlar el meta nivel desde el nivel base.

2.3 Asociación de Meta Objetos con las Entidades del Nivel Base

Los meta objetos pueden asociarse con objetos, clases y referencias.

La asociación del meta objeto con una entidad del nivel base significa que todos los eventos involucrados en la ejecución de esa entidad base son enviados al meta objeto asociado. Si el meta objeto está asociado a una clase, todos los eventos generados por cualquier instancia de esa clase son enviados al meta objeto. Si está asociado a una referencia, solo se envían al meta objeto los eventos generados por operaciones que utilizan esta referencia.

La asociación es explícita, mediante las operaciones attachObject, attachReference y attachClass.

No todo objeto debe tener un metaobjeto asociado a él. Los meta objetos pueden asociarse dinámicamente con los objetos del nivel base en tiempo de ejecución, lo que significa que las aplicaciones solo sufren la sobrecarga del meta sistema si lo necesitan.

2.4 Estructura

La extensión MetaJava consiste en dos capas. La capa inferior abre la JVM y recibe el nombre de **Interfaz del Meta Nivel**, del inglés, *meta-level interface*. La capa superior está escrita en Java y proporciona el mecanismo de eventos descrito, utilizando generación de código en tiempo de ejecución.

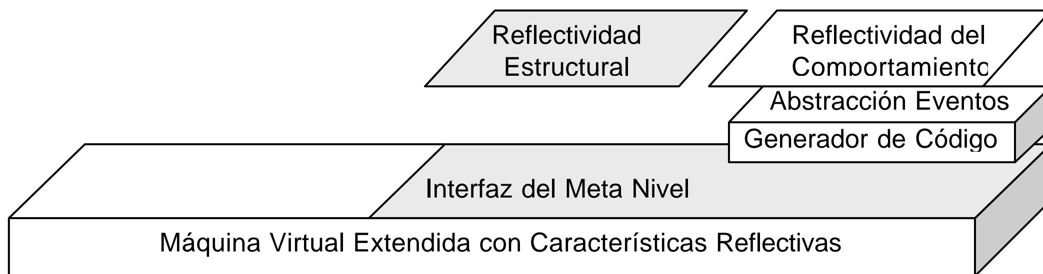


Figura 13.3 Estructura de MetaJava

2.4.1 Meta-level interface o MLI

Las principales funciones de la MLI se agrupan en los siguientes apartados.

- **Manipulación de datos del objeto**. El propósito es modificar el estado de un objeto. Se trata de funciones para acceder a las variables instancia de objetos y para acceder a los nombres y tipos de las mismas.
- **Soporte para modificación del código**. Se trata de funciones para recuperar los bytecodes de un método y modificarlos.
- **Enlace nivel base-meta nivel**. Interfaz de bajo nivel para el enlace entre el nivel-base y el meta-nivel que los meta objetos usan para implementar la asignación.
- **Modificación de la estructura de clase**. Los meta objetos necesitan cambiar la semántica de objetos o referencias, para lo que modifican las estructuras de la clase.
- **Modificación del pool de constantes**. Debe cambiar si cambian los métodos de una clase.
- **Ejecución**. Este grupo contiene funciones que posibilitan la ejecución de métodos arbitrarios.
- **Creación de instancias**.

2.5 Reflectividad del Comportamiento

La reflectividad del comportamiento se consigue asociando meta objetos con los objetos del nivel base y transfiriéndoles el control ante determinados eventos.

Por tanto, la implantación de la reflectividad del comportamiento implica dos mecanismos: **establecer la relación entre objetos y meta objetos** e **implantar un sistema de eventos**.

1.1 Implantación de la Relación objetos base – meta objetos: Shadow Classes

Para cambiar la semántica de un objeto sin afectar la semántica de otros objetos de la misma clase, el sistema MetaJava utiliza el concepto de clase como contenedor de toda la información acerca de un objeto, incluida la información relativa a su comportamiento. Para ello implanta la abstracción de **shadow classes**.

Una **shadow class** C' es una copia exacta de la clase C con las siguientes propiedades:

- C y C' son indistinguibles desde el nivel base
- C' es idéntica a C excepto por modificaciones hechas por un meta-programa
- Los campos y métodos estáticos se comparten entre C y C'.

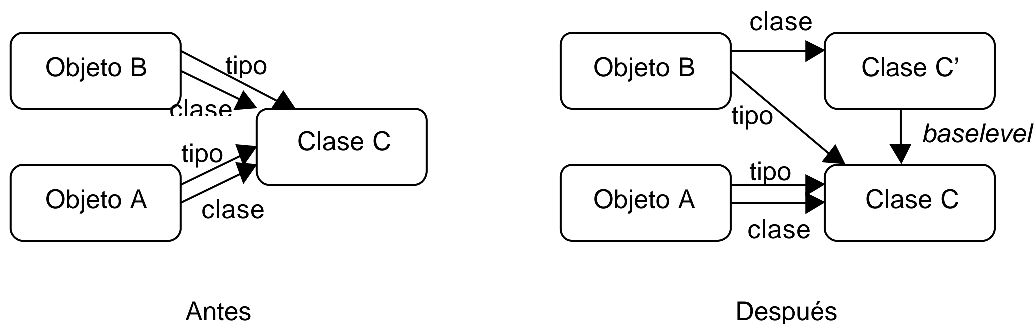


Figura 13.4 Creación de una shadow class

Los lenguajes basados en clases suponen, normalmente, que existen muchos objetos del mismo tipo (clase). Los lenguajes basados en prototipos como Self [Cha92] o Moostrap soportan una especificación de reusabilidad distinta. Definen objetos *one-of-a-kind* [MMC95], donde es fácil derivar un objeto de otro y cambiar los campos o métodos del nuevo objeto, sin afectar al original. Las shadow clases de MetaJava son una aproximación a este comportamiento.

2.5.1 Asociación de Meta Objetos a los Objetos del Nivel Base

La Meta Java Virtual Machine (MJVM), almacena los objetos a través de asas de objetos, del inglés *handles*, que contienen punteros a los datos del objeto y a la clase del objeto.

Cuando se asocia un meta objeto con un objeto de nivel base, se crea una shadow class y el enlace del objeto a su clase se redirecciona a la shadow class, a la vez que se asocia el meta objeto con la shadow class mediante el enlace **metaobject**.

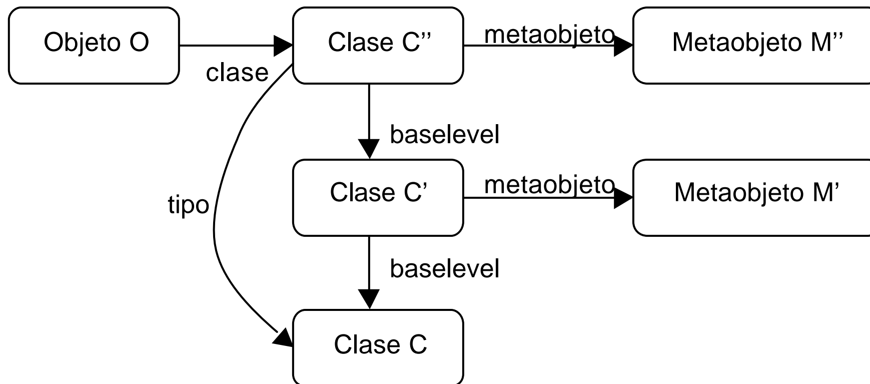


Figura 13.5 Enlace de objeto y metaobjeto: Torre de meta objetos

2.5.2 Torre Reflectiva

En MJVM, cada clase establece enlaces con el metaobjeto que tenga asociado, si tiene alguno, y con la clase base, si existe.

- **Enlace de tipo**. Cuando se crea un objeto, el enlace de tipo se establece automáticamente a la clase original, mientras el enlace de clase se establece a la clase correspondiente, C o C’.
- **Enlace baselevel**. Cada shadow class tiene un enlace a su clase base.

La torre reflectiva en MJVM se construye siguiendo el enlace *baselevel*, que implanta una torre de meta objetos. Ver Figura 13.5.

2.5.3 Problemas de la Implantación

La utilización de shadow classes introduce algunos problemas:

- **Mantenimiento de la Consistencia entre C y C’**. Todos los datos no constantes relativos a la clase deben compartirse entre C y C’, dado que las shadow classes no deberían cambiar las clases sino propiedades de los objetos.
- **Recolección de Shadow classes** cuando se desasocian del objeto.
- Algunos hilos pueden ejecutarse en un objeto base de clase C cuando se está creando y modificando la shadow class. El sistema debe garantizar que el código original se mantiene y usa al menos, mientras se ejecute el código.

2.6 Transferencia del Nivel Base al Meta Nivel: Eventos

El sistema de generación de eventos es una capa sobre el MLI que proporciona la abstracción de reflexión del comportamiento.

La capa de generación de eventos para eventos locales se implementa completamente en términos de las facilidades de reflectividad estructural.

Si el meta objeto está interesado en un comportamiento específico del objeto del nivel base, registra el evento con uno de los métodos definidos al efecto.

Exposición de la recepción y ejecución de mensajes

Se ejecuta un método del meta-nivel antes de que se ejecute el método del nivel base y otro método del meta-nivel después.

Estos métodos se seleccionan de acuerdo al atributo methodclass del método del nivel base. El nombre del método del gestor de eventos responsable en el meta-nivel es la concatenación del atributo methodclass y Entry para los métodos a ejecutar antes y Exit para los métodos a ejecutar después.

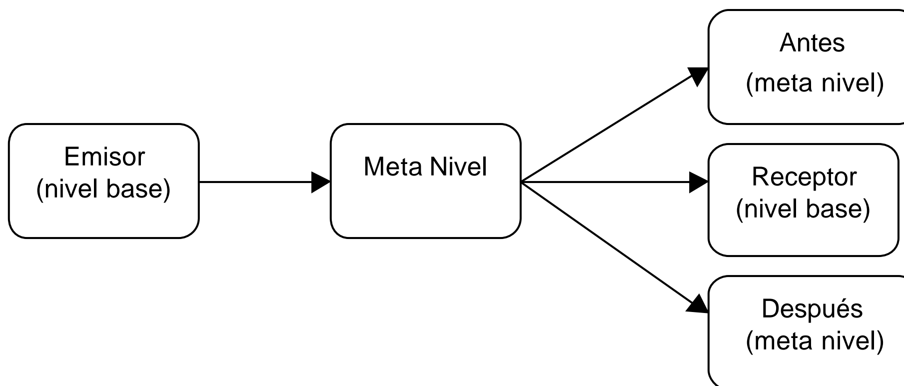


Figura 13.6 Notificación de la invocación de un método (antes, después)

El esquema de notificación antes/después permite a metaobjetos individuales verificar algunas restricciones antes o después de la ejecución de un método o retrasar la misma.

Para recibir eventos de la invocación de un método, el meta objeto debe registrarse en la capa de gestión de eventos utilizando el método void registerEventMethodCall (Object base-obj, int notificationType, String methods[]).

Exposición del envío de mensajes

La invocación de métodos ejecutada por el nivel base es otro de los eventos que pueden provocar la transferencia de control del nivel base al meta nivel. Para recibir un evento para una invocación a un método, el meta objeto debe registrarse también en la capa de gestión de eventos utilizando el método: void registerEventMethodCallOut (Object base-obj, String methods[]).

Exposición del acceso a las variables

El acceso a las variables instancia de los objetos del nivel base también se notifica de forma similar. El meta objeto puede instalar gestores para los accesos de lectura y escritura. El gestor de escritura se invoca cuando se va a escribir la variable y el de lectura cuando se intente leer, y retorna el valor. Igualmente, el meta objeto debe registrar el evento: void registerEventFieldAccess (Object Base-obj, String fields[]).

Exposición del bloqueo

Para recibir un evento si se bloquea un objeto del nivel base el metaobjeto debe utilizar: void registerEventObjectLock (Object base-obj).

Exposición de la creación de objetos

Para recibir un evento si se crea un objeto de una clase clazz, el meta objeto debe invocar: void registerEventObjectCreation (Class clazz).

Exposición de la Carga de Clases

Para recibir un evento si se intenta cargar la clase classname, el meta objeto debe invocar: void registerEventClassLoading (String classname).

La implantación suele llevarse a cabo reemplazando los bytecodes con código stub que salta al meta espacio.

3 DSOM

SOM es una tecnología Orientada a Objetos nacida con el objetivo de establecer un mecanismo que permita la interoperabilidad entre objetos independientemente del lenguaje de programación o el sistema operativo en que se hayan definido. Para lograrlo, SOM establece un modo de funcionamiento que gira alrededor del concepto de **interfaz e implantación de clases**.

Una clase está caracterizada por una **interfaz** o conjunto de métodos que determinan su comportamiento. A partir de ellas, pueden crearse instancias denominadas **objetos** que responderán a invocaciones hechas a cualquier método definido en la interfaz de la clase a la que pertenezca.

La interfaz de una clase estará implementada por un conjunto de procedimientos que se ejecutarán como respuesta a la invocación de los métodos de la misma. Sin embargo, este hecho es completamente transparente para los clientes, que sólo conocerán los servicios ofrecidos por el objeto servidor.

Para ocultar los detalles de **implementación**, SOM establece una estricta separación entre la definición de la interfaz de una clase y su implantación, es decir, entre los métodos que soporta esa clase y los procedimientos que implementan estos métodos.

Gracias a esto, los clientes son totalmente inmunes a cambios en detalles de implementación de los procedimientos que implementan los métodos de la clase que utilizan, no necesitando recompilación ni recarga ante estos cambios. Además, SOM permite la construcción, empaquetamiento y distribución de librerías binarias de clases suministrándoselas a aplicaciones clientes que podrán utilizar las clases allí contenidas sin ninguna restricción derivada del lenguaje o el sistema operativo en el que fueron escritas.

SOM proporcionará a estos clientes herramientas necesarias para poder crear objetos instancias de una clase, construir a partir de ellas nuevas clases aplicando la técnica de la herencia y formando así una jerarquía de clases, o invocar métodos en objetos. SOM dispone de un entorno en tiempo de ejecución que permite la realización de todas estas operaciones.

Así pues SOM es un estándar binario para objetos, neutral respecto al lenguaje y al sistema operativo, concebido para resolver el problema del fuerte acoplamiento binario entre una aplicación y las bibliotecas de clases que ésta utiliza.

Otros objetivos más ambiciosos como permitir la operación y comunicación entre objetos situados en diferentes espacios de direcciones o incluso en diferentes máquinas quedan fuera, inicialmente de las metas marcadas por SOM. Sin embargo, en los últimos años, ha sufrido una profunda evolución para lograr este tipo de interoperabilidad entre objetos, desarrollándose, sobre SOM una herramienta denominada DSOM. Con esta evolución, ambas han quedado íntimamente ligadas, sin que sea posible ya diferenciar cuando se habla de SOM a qué se está refiriendo.

3.1 Modelo de objetos de SOM

El modelo de objetos de SOM es un modelo avanzado, donde las clases definen las características de los objetos que son entidades en tiempo de ejecución con un conjunto específico de métodos y variables instancia.

Asociada con cada clase SOM, existe una tabla de métodos denominada **tabla de métodos de las instancias**. Esta tabla es compartida entre todos los objetos instancia de esa clase y contiene punteros a los procedimientos que implementan los métodos soportados.

El modelo de objetos de SOM proporciona encapsulación, herencia múltiple con mecanismos para resolver ambigüedades, polimorfismo, metaclases, diferentes modos de resolución de métodos y creación dinámica de clases. Además SOM proporciona una jerarquía básica de clases a partir de las cuales se generan el resto de clases y metaclases.

Clases: Objetos de primera clase

Las clases SOM se representan como objetos en tiempo de ejecución llamados **objetos clase**, a diferencia de la mayoría de los lenguajes de programación orientada a objetos para los cuales las clases son estructuras con significado sólo en tiempo de compilación. De esto se derivan algunas ventajas entre las que destaca la posibilidad de establecer o acceder a la información referida a una clase en tiempo de ejecución.

3.2 Reflectividad en DSOM : Metaclases

Al igual que los objetos, **las clases son entidades en tiempo de ejecución**, es decir, objetos instancias de las metaclases de SOM. La clase que define la implantación de objetos clase se denomina **metaclase** y, al igual que los objetos instancia son instancias de una clase, los objetos clase son instancias de una metaclase.

Las metaclases de una clase definen los métodos de la clase, es decir, los métodos que una clase puede ejecutar. Estos métodos también reciben el nombre de **factory methods** o constructores y llevan a cabo tareas como crear nuevas instancias de una clase, mantener cuenta del número de instancias de la clase, etc. Además, los métodos de la clase facilitan la herencia de los métodos de la instancia desde las clases padre.

3.3 Jerarquía de Objetos SOM

La jerarquía de objetos SOM es una estructura en forma de árbol con una única raíz. El sistema SOM ofrece tres clases primitivas, que son la base para el resto de las clases a partir de las cuales se derivan el resto.

3.3.1 Objeto Class SOMObject

Es la clase raíz de todas las clases SOM y, por tanto, las demás clases que se definan en el sistema derivarán directa o indirectamente de ésta. Por esta razón, los métodos proporcionados por esta clase son heredados por todas las demás y definen el comportamiento común a todos los objetos SOM.

3.3.2 Objeto Class SOMClass

SOMClass es la clase padre de todas las metaclases de SOM y, al igual que ocurría en SOMObject, los métodos de SOMClass definen el comportamiento básico de todos los objetos clase SOM, como por ejemplo el mecanismo a usar para resolver los conflictos de la herencia.

SOMClass proporciona métodos de clase para crear nuevas instancias y métodos de clases para obtener o actualizar dinámicamente información acerca de una clase y sus métodos en tiempo de ejecución.

Por ser un objeto SOM, SOMClass deriva de la clase básica SOMObject, y por tanto los objetos clase de SOM heredan el conjunto de métodos básico proporcionado por el objeto clase SOMObject.

3.3.3 Objeto SOMClassMgr Class y SOMClassMgrObject

SOMClassMgrObject actúa de gestor de la información acerca de las clases. Mantiene un registro o directorio en tiempo de ejecución de todas las clases SOM en el proceso actual y se encarga de la carga y descarga dinámica de las clases localizadas en librerías de clase.

La siguiente figura ilustra las relaciones entre las clases, metaclases y objetos en el tiempo de ejecución SOM.

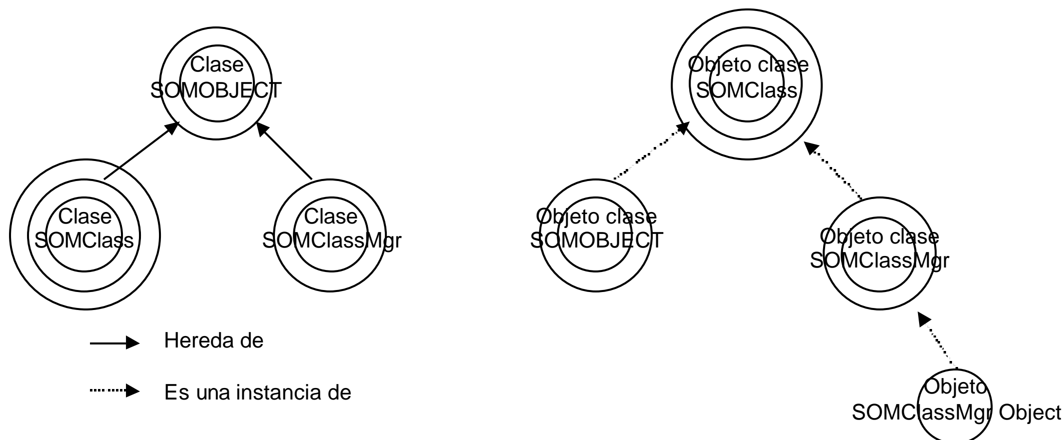


Figura 13.7 Jerarquía básica de clases SOM

3.4 Introspección de Objetos

La utilización de objetos clase permite utilizar las clases como si fuesen instancias, que lo son, pudiendo así conseguir la clase de un objeto, crear un nuevo objeto clase o referirse a un objeto clase a través de un puntero.

3.4.1 Clase de un objeto

SOM implementa el método somGetClass en los objetos SOM que devuelve un puntero al objeto clase del que el parámetro es instancia. Conseguir la clase de un objeto es útil para realizar **introspección** acerca del objeto.

Una clase derivada puede sobreescribir este método para proporcionar una semántica alternativa y, supuestamente, mejorada para un objeto.

A partir del objeto clase, se puede inspeccionar la representación en tiempo de ejecución en múltiples aspectos. Así, en un objeto clase podemos invocar, entre otros, métodos como somGetInstanceSize que calcula el tamaño, en bytes, necesario para una instancia de una clase, somGetParents que obtiene la secuencia de las clases padres o somSupportsMethod que determina si una instancia de una clase soporta un método determinado.

3.4.2 Clase Objeto

Además de utilizar el objeto clase, SOM también define un conjunto de métodos que pueden invocarse en cualquier objeto SOM.

Estos métodos tienen como finalidad inspeccionar la representación del objeto en tiempo de ejecución y conseguir información como:

- La clase del objeto, nombre u objeto clase
- El tamaño que ocupa la instancia
- Determinar si un objeto es una instancia de una clase dada o de una de sus clases descendientes.
- Si es una instancia de una clase determinada.
- Si responde a un método determinado

Además de estos, la clase SOMObject, introduce métodos que permiten la invocación dinámica de métodos en un objeto, permitiendo la llamada a métodos cuyo nombre o argumentos se desconocen en tiempo de compilación.

4 Smalltalk

Smalltalk[GR83] exhibe muchas características reflectivas. **Las clases son objetos de primera instancia**, instancias de **metaclases** que determinan su comportamiento, permite que los programas examinen su entorno en tiempo de ejecución y puedan redefinir sus métodos.

La reflectividad en Smalltalk, basada en proporcionar métodos que permitan el acceso a información en tiempo de ejecución del objeto y en exponer tal información al exterior como objetos, es, fundamentalmente, **reflectividad estructural**.

4.1 Meta Objetos en Smalltalk

Entendiendo un **meta objeto**, tal y como se define en [KRB91], como un **objeto que representa fragmentos de un programa**, en Smalltalk, los objetos normales se

utilizan para modelar el mundo real y los meta objetos describen estos objetos ordinarios.

Es decir, los meta objetos describen las entidades de Smalltalk. Así, Smalltalk ofrece clases de meta objetos para definir:

- La estructura de una entidad: Class, Behavior, MetaClass
- La semántica: Compiler, Parser
- El comportamiento: Message, CompiledMethod
- Control de estado: Context, Process, ProcessorScheduler
- Y otros aspectos.

4.2 Aspectos Reflectivos en Smalltalk

Los aspectos reflectivos más importantes son:

- Las **meta operaciones**: objetos como meta objetos
- La **estructura**: clases como objetos
- La **semántica**: compiladores como objetos
- **Envío de mensajes**: mensajes como objetos
- **Control**: procesos como objetos

4.2.1 Meta Operaciones

[LP90] define las **meta operaciones** como **operaciones que proporcionan información acerca de un objeto**, en contraposición de información contenida en un objeto.

Modelo

Las principales meta operaciones, definidas en Smalltalk como métodos de la clase object, raíz de la jerarquía de herencia se refieren a:

- **Estructura interna de un objeto**. Permite leer y escribir una variable de instancia mediante un índice, en lugar de utilizar el nombre de la variable
- **Meta representación de un objeto**. Permite no sólo identificar la clase de un objeto, sino incluso cambiarla modificando, por tanto, el comportamiento del objeto.
- **Identidad de un objeto**. Permite identificar a todos los objetos que están referenciando a uno dado.

Uso

Estas facilidades permiten la construcción de una clase inspector, que implanta la introspección.

4.2.2 Estructura

La reflectividad estructural implica que el lenguaje sea capaz de proporcionar una exposición completa tanto del programa en ejecución, como de sus datos. Smalltalk manipula únicamente objetos, instancias de una clase que describe no sólo su comportamiento sino también su estructura.

Implanta la clase object, que define el comportamiento básico de todo objeto del sistema.

Modelo

Las clases, como objetos regulares, se describen mediante otras clases llamadas **metaclases**. La herencia de metaclases sigue el mismo esquema que la herencia de clases de nivel base

El núcleo de clases y meta clases de Smalltalk se define con cinco clases:

- **Object**. Define el comportamiento por defecto para todos los objetos.
- **Behavior**. Define el comportamiento mínimo para las clases, especialmente su representación.
- **ClassDescription**. Se encarga de describir el comportamiento común de las clases y metaclases tal como organización de los métodos, y mecanismo para persistir.
- **Class**. Se refiere al comportamiento de las clases.
- **Metaclass**. Describe el comportamiento de las metaclases.

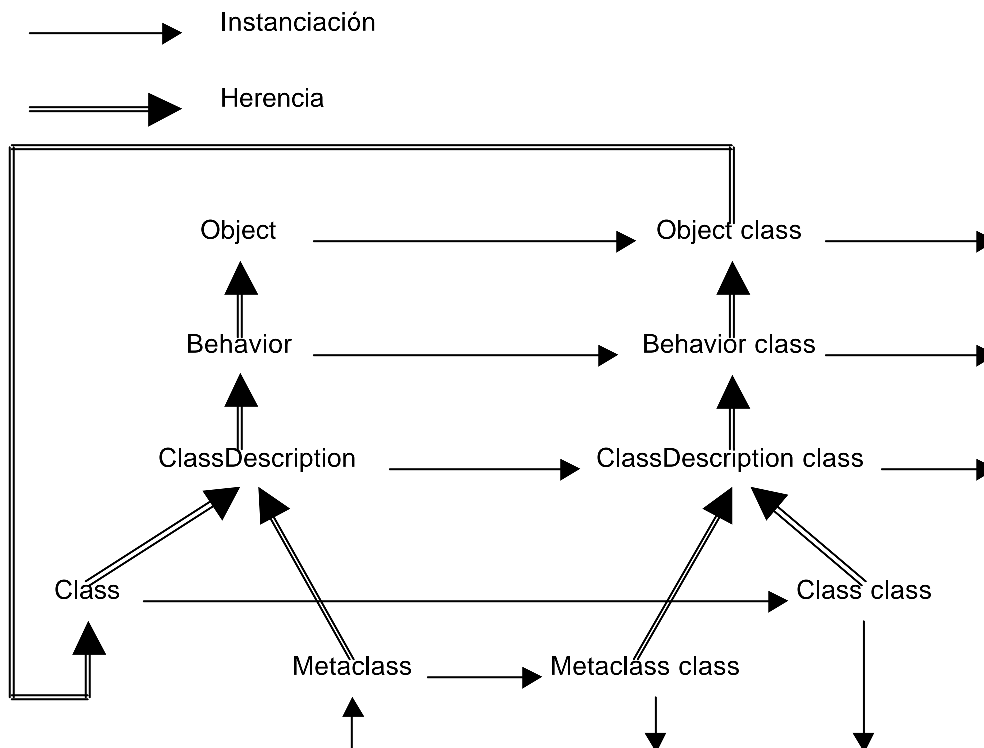


Figura 13.8 Núcleo de Clases/MetaClases en Smalltalk

Uso

La exposición de clases permite al lenguaje proporcionar utilidades esenciales como *implementors* (inspectores de clases en busca del método que encaja con un nombre), *senders* (inspeccionan los métodos en busca de aquel que ejecute un mensaje enviado) y *messages* (buscan implementadores de un mensaje presente en un método).

4.2.3 Envío de Mensajes

Modelo

En Smalltalk, todo comienza en el envío de mensajes, que se compone de dos fases, lookup, búsqueda del método de acuerdo al mensaje recibido, y apply, aplicación del método.

Lookup tiene lugar en tiempo de ejecución y utiliza la información de las clases, con lo que, toda la información necesaria es accesible y modificable desde el lenguaje: métodos, diccionarios de métodos, superclases, comportamiento, etc., se representan como objetos.

Los mensajes se exponen mediante la utilización de los mensajes como objetos de primera clase, instancias de la clase Message. Pero esta exposición sólo se produce en el caso particular de que el lookup falle, es decir, cuando la máquina virtual envíe el método doesNotUnderstand al receptor original, con un objeto mensaje que expone el mensaje enviado, como argumento.

Uso

La gestión del fallo en la etapa de lookup, permite la construcción de un mecanismo similar a la implementación de mensajes asíncronos en ACTALK[Bri89], mediante la especialización del método doesNotUnderstand.

4.2.4 Control de Estado

El sistema Smalltalk está basado en la exposición de procesos y, en general, en la exposición de los objetos necesarios para construir un sistema con multiprocesamiento. Los procesos gestionan la planificación de tiempo (timingPriority), eventos de entrada (lowIOPriority) y eventos de usuario.

Modelo

Processor, la única instancia de la clase ProcessorScheduler, coordina el uso del procesador físico por parte de todos los procesos que requieren servicio. Define una semántica requisable entre procesos que tienen distintas prioridades.

Processor yield permite la cesión del procesador, la clase semaphore proporciona comunicación sincronizada entre procesos y la clase delay representa un retraso real de tiempo en la ejecución de un proceso lo que proporciona características de planificación en tiempo real.

La característica reflectiva más reseñable de Smalltalk es la exposición de la pila en tiempo de ejecución de cualquier proceso, a través de una cadena de capas de la pila, denominadas contexts. La variable thisContext devuelve el contexto actual del proceso en ejecución.

Un contexto fundamentalmente mantiene la siguiente información:

- El contexto (emisor) que lo ha creado vía la invocación de un método.
- El método que se está ejecutando
- Un puntero de instrucción
- El receptor del mensaje y sus argumentos.

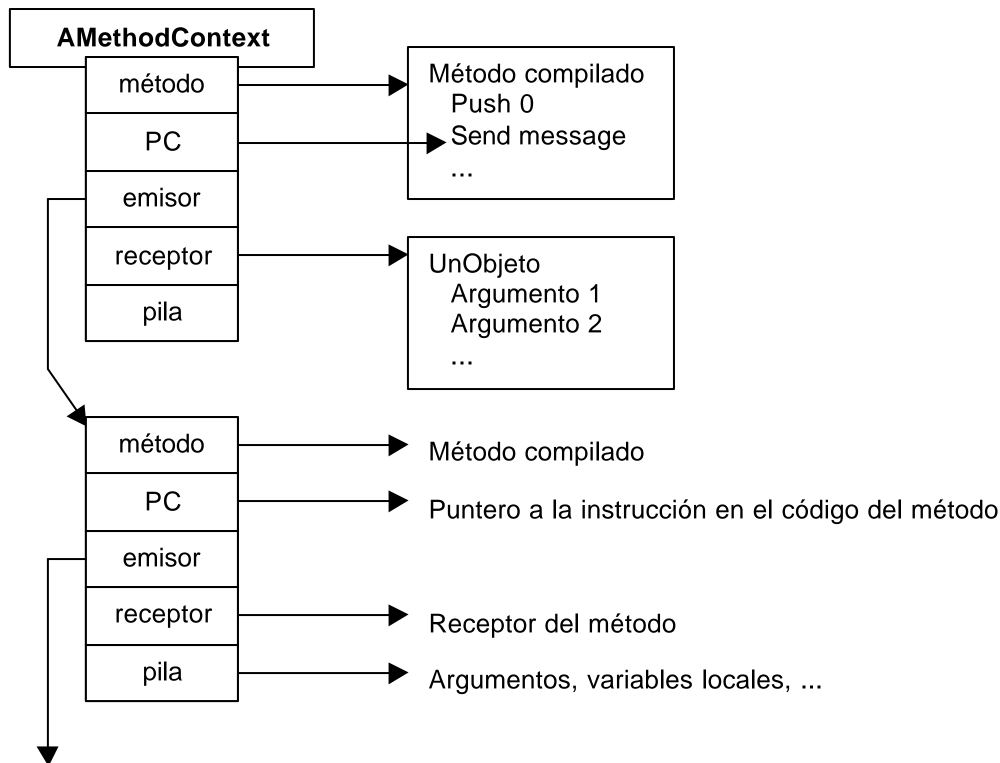


Figura 13.8 Pila de ejecución

Uso

Smalltalk permite a los programas controlar completamente su propia ejecución utilizando objetos regulares como Context. Es lo que se denomina **Intercesión**.

Dos aplicaciones directas de este control en la ejecución son la implantación de un mecanismo de gestión de excepciones en Smalltalk, que modifica el esquema regular de ejecución y la construcción de una herramienta de depurado.

- **Gestión de Excepciones**. La clase Exception expone objetos que manipulan la pila de ejecución con el fin de gestionar errores. Las excepciones se lanzan a través de la pila, y son capturadas por gestores que definen el mensaje handle: do:, con el fin de realizar las acciones apropiadas ante errores. Esta implantación puede extenderse o reemplazarse para proponer un sistema de gestión de errores alternativo para Smalltalk[Don90].
- **Depuración**. Una herramienta de depuración construida sobre este mecanismo reflectivo podría consultar cualquier contexto de la pila de ejecución, determinar qué parte del contexto seleccionado está siendo ejecutada, inspeccionar el receptor del mensaje o los argumentos e incluso modificar cualquier contexto recompilando su método y continuar su ejecución con el nuevo código.

Capítulo XIV

Integración de Máquina Abstracta + Sistema Operativo: Añadir Reflectividad al Sistema

Como se sugirió en el Capítulo II y se detalló en el Capítulo XII, la reflectividad es una arquitectura que da como resultado un sistema flexible.

Adoptada en esta tesis como medio para integrar la Máquina Abstracta Orientada a Objeto y el Sistema Operativo Orientado a Objeto, se detalla en este capítulo la arquitectura resultante.

1 Diseño de un Entorno de Computación Orientado a Objeto

La idea básica, compartida con trabajos como CodA[McA95a], Apertos[Yok92], Merlin[AK95] y otros, es proporcionar un entorno de computación OO flexible que dé soporte a la abstracción de objeto como elemento que encapsula datos y ofrece una interfaz o conjunto de métodos para la manipulación de tales datos, que se ejecutan ante la invocación por parte de otros objetos, o incluso de él mismo.

Como primer paso para el diseño de un entorno de computación OO, se estudiaron en los capítulos VIII y IX el conjunto de restricciones funcionales y no funcionales a establecer a la hora de implantar el mecanismo de computación de un sistema operativo orientado a objetos.

Posteriormente, en el capítulo X, se discutieron dos modelos de objetos desde el punto de vista de los requisitos establecidos, haciendo especial hincapié en las diferencias entre el **Modelo de Objetos Activo** y el **Modelo de Objetos Pasivo**, así como en las ventajas e inconvenientes de cada uno, para finalmente, establecer los indudables beneficios del primero respecto al segundo.

De todo ello, se derivan una serie de principios que deben mantenerse a lo largo del diseño y que llevan a definir claramente el mecanismo de computación implementado para un Sistema Operativo Orientado a Objetos[1].

1 Aunque todos ello fue ampliamente discutido en los capítulos citados anteriormente, se citarán brevemente a continuación con el fin de facilitar la lectura del resto del Capítulo.

1.1.1 Requisitos Funcionales

Brevemente, se establecen sobre el entorno de computación, un conjunto de requisitos funcionales, de los que dependerá, en gran parte, la sobrecarga de aquellas aplicaciones que quieran hacer uso de él.

Abstracción Uniforme

Debe dar soporte al objeto como entidad en sí, y no como elemento que vive completamente dentro de un espacio de direcciones de un proceso. Tal abstracción debe ser adecuada a los requerimientos de niveles superiores.

Todos los objetos del sistema deben ser iguales, con un conjunto de propiedades y mecanismos estándar, aunque puedan implementarse distintas políticas sobre tales mecanismos, en base a la aplicación de uno de los principios de diseño más importantes en los Sistemas Operativos: **Separación de Política y Mecanismo**[Cro96].

Entorno de Ejecución Concurrente

La abstracción de objeto, que ofrece servicios a otros objetos en el sistema, emula en cierta medida a los servidores multihilo que aparecieron y se desarrollaron dentro del paradigma cliente/servidor.

La posibilidad de que el sistema dé soporte a una abstracción similar a los servidores, objetos multihilo capaces de servir varias peticiones simultáneamente, se presenta como un camino prometedor para construir un entorno concurrente que no imponga a los objetos restricciones a priori, que mengüen su rendimiento[WY88].

Comunicación Uniforme

Siendo los objetos entidades autónomas, que encapsulan su estado y con mecanismos que le permitan ejecutar sus métodos, es necesario hacer posible la comunicación entre objetos, de forma que puedan solicitar de otros, tareas que no puedan llevar a cabo por sí mismos.

Control de la actividad interna y Planificación.

El objeto debe contar con mecanismos internos que le permitan, no sólo establecer el grado de concurrencia interna deseada o recomendable[YTY+89] (un único método, múltiples métodos distintos, múltiples instancias de un mismo método...), sino también establecer un control más fino sobre la ejecución concurrente de distintos métodos.

Es decir, debe ser capaz de suspender y reanudar la ejecución de sus métodos, atendiendo a factores como el riesgo en la consistencia de su estado interno, el balance de carga, etc.

De la misma forma, ante un conjunto de métodos ejecutables, debe estar dotado de los mecanismos necesarios para aplicar una política que le permita establecer turnos de ejecución según criterios como las prioridades del método, del solicitante, u otras que el cliente o desarrollador de la aplicación consideren oportunas.

Trabajos como Apertos[Yok92], CodA[McA95a], Java//[CV98], MetaJava[GK97b] y otros, han estudiado este problema con distintos resultados.

Así, por ejemplo, Apertos serializa las actividades en un objeto definiendolo como una entidad serie de ejecución. Por su parte, CodA[McA95a] permite definir el modelo

que sigue el objeto para la ejecución al exponer los protocolos que definen la recepción y ejecución de mensajes.

Excepciones

Por último, el sistema debe estar dotado de un mecanismo de excepciones que permita gestionar de forma adecuada los problemas que se produzcan, sin que ello deba suponer en todo caso, la terminación errónea de la aplicación.

1.1.2 Requisitos No Funcionales

Tanta o mayor importancia que los requerimientos funcionales, tiene el establecimiento de un conjunto de requerimientos no funcionales, y el diseño de un sistema que se adapte a tales normas.

En este sentido, en el Capítulo IX se lleva a cabo una detallada discusión de cuáles deberían ser los requerimientos no funcionales a tener en cuenta en un sistema como el que pretendemos diseñar.

Abstracción única

El soporte del entorno de computación a la ejecución, planificación y comunicación de los objetos no debería suponer la adición de nuevas abstracciones con semántica distinta a la definida para el objeto.

Se hace pues necesario desarrollar un objeto que abarque datos, métodos y que encapsule a su vez la computación y los mecanismos necesarios para gestionarla, evitando la división de objetos entre objetos con y sin capacidad de ejecución, lo que deriva nuevamente en economía de conceptos y, por tanto, sencillez conceptual.

Flexibilidad

La funcionalidad solicitada por los objetos de usuario, y ofrecida por el entorno de computación, debería ofrecerse a los objetos de las aplicaciones de forma que, cada objeto o conjunto de objetos relacionados, pudiese adaptar su entorno de ejecución con el fin de conseguir la máxima flexibilidad.

Permitiendo variaciones en los mecanismos ofrecidos, es posible salvar la distancia conceptual entre la funcionalidad requerida y la ofertada o *mapping dilemmas* [KLM+93].

1.1.3 El modelo de objetos

En el **modelo de objetos activo**, los objetos son entidades auto-contenidas, dotadas de un protocolo de comunicación unificado. Cada objeto se comporta como un agente autónomo que puede comunicarse con otros objetos conocidos[Fer88, MHM+95].

En este marco orientado a objetos, la computación se representa como una secuencia de paso de mensajes entre objetos y supone la petición de servicio por parte del objeto origen de la invocación al objeto destino de la misma. Este último lleva a cabo las acciones indicadas por los mensajes solicitados.

La definición de objeto propuesta en esta tesis, puede llevar a cabo varias acciones simultáneamente.

Cada objeto puede recibir varios mensajes y decidir procesarlos inmediatamente o posponerlos hasta que se den ciertas condiciones idóneas o necesarias para su ejecución. Por tanto, los objetos estarán dotados de capacidad plena para gestionar la ejecución de sus métodos.

Dado que cada objeto es una entidad auto-contenida provista de un mecanismo de comunicación, la descomposición de un sistema en una colección de objetos es muy flexible y la estructura del sistema resultante es muy natural.

Es más, el mecanismo de comunicación uniforme y la auto-contención protegen a los objetos de accesos ilegítimos, estableciendo políticas de interacción ordenadas entre objetos [YT88a].

1.2 Entorno de Computación = Sistema Operativo + Máquina abstracta

Habitualmente, las aplicaciones requieren, más o menos, la misma funcionalidad del entorno de computación. Sin embargo, no todas ellas la requieren de la misma forma. Por ejemplo, en un entorno de computación OO, todos los objetos necesitarán un mecanismo que les permita comunicarse con otros objetos.

Sin embargo, es muy posible que no todos ellos requieran una comunicación igual. Algunos objetos pueden tener más importancia o privilegios, y les estará permitido comunicarse con objetos fuera de aquellos definidos por la misma aplicación, o incluso con objetos remotos, situados en distintas máquinas.

Otros objetos, sin embargo, serán objetos auxiliares o de carácter temporal o local a la aplicación, que sólo ven y son vistos por objetos dentro de su propia aplicación.

Aunque el mecanismo de comunicación es necesariamente el mismo, el paso de mensajes, las características de la comunicación, la implantación concreta del mecanismo que se necesita en cada uno de los casos, es distinta.

La flexibilidad es, de los requerimientos no funcionales, uno de los que más peso va a tener a la hora de diseñar el sistema de computación.

1.2.1 Carbayonia + SO4

Para proporcionar toda la funcionalidad requerida para un sistema de computación OO, a la vez que se cumplen los requisitos de unicidad en la abstracción, flexibilidad y otros, la solución propuesta en esta tesis, es la **integración de una máquina abstracta OO** que ofrezca el soporte básico para la computación y **un SOOO** que ofrece el resto de la funcionalidad. En los Capítulos XV y XVI se ofrece una descripción detallada de cada uno de ellos.

La máquina abstracta ofrece la funcionalidad básica, aquella que resultaría caro, peligroso o difícil implantar en cualquier otra capa del Sistema Integral.

Por su parte, la funcionalidad ofrecida por el Sistema Operativo extenderá el nivel básico, es decir, añadirá funcionalidad a aquella que tenga codificada internamente la máquina, o bien adaptará el comportamiento de la máquina abstracta modificando las funciones que ofrece.

Extensión Uniforme con el Resto del Sistema

El Sistema Operativo ofrece su funcionalidad mediante objetos no diferenciables de los objetos definidos por los usuarios y que se invocan de forma similar (**extensión uniforme con el resto del sistema**) ofreciendo un entorno similar al que ofrecen las aplicaciones de usuario.

Esta opción (máquina abstracta OO + sistema operativo OO que extiende el comportamiento de la primera) tiene varias ventajas respecto a otras posibilidades como la integración de todas las características en la máquina, como son la flexibilidad mientras que la segunda resulta, por el contrario, más eficiente.

2 Entorno Flexible = Reflectividad (Sistema Operativo + Máquina abstracta)

Los sistemas de computación tradicionales consisten en un sistema operativo y un programa que explota los servicios del sistema operativo utilizando una interfaz para la programación de aplicaciones (API).

La aproximación reflectiva que aquí se propone para la organización del entorno de computación integral OO es distinta.

La estructura propuesta gira alrededor de una **máquina abstracta**, nivel básico que ofrece la funcionalidad mínima necesaria para el soporte a la existencia de los objetos, la ejecución de sus métodos y su comunicación. En este aspecto, se asemeja a un exokernel[EKT94], minimizando la funcionalidad implementada en la máquina para hacer posible la flexibilidad.

Por encima de ella, el **sistema operativo** extiende la funcionalidad básica por medio de un conjunto de objetos hasta proporcionar un entorno de computación adecuado. Esta es la principal diferencia con los exokernels. Mientras estos se basan en complejos mecanismos para permitir incorporar código al exonúcleo, la reflectividad permite la extensión uniforme y sencilla de la máquina, mediante objetos.

Es decir, mientras la máquina ofrece los mecanismos más básicos, el sistema operativo complementa estos mecanismos para ofrecer un sistema de gestión de objetos en el que estos puedan ejecutar sus métodos concurrentemente, controlándose su ejecución mediante algún tipo de política, así como comunicarse con otros objetos.

Haciendo una analogía con los sistemas tradicionales, se podría decir que la máquina abstracta ofrece los mecanismos mientras el sistema operativo ofrece la política.

La necesaria integración de ambos niveles puede llevarse a cabo aplicando varios principios, como queda reflejado en el capítulo II. Sin embargo, después de profundizar en la **reflectividad** en el capítulo XII, esta se ha mostrado como el mecanismo más prometedor para obtener un entorno de computación flexible de la suma de una máquina abstracta y un sistema operativo.

2.1 Reflectividad: Arquitectura Genérica

El concepto fundamental en el diseño del sistema de computación será la **división del espacio de objetos** en dos niveles, el **nivel base** y el **meta-nivel** o nivel de apoyo para el nivel base.

Arquitectura del MetaNivel

La clave en el diseño de la arquitectura del meta-nivel propuesta en esta tesis es la **descomposición** o **factorización** del mismo en objetos de grano fino **o meta-componentes** o **meta-objetos**. De esta forma se mantiene la **uniformidad conceptual en torno al paradigma de la Orientación a Objetos** en todo el sistema.

Se puede establecer una analogía clara con la programación tradicional. Cuando se desarrollan o programan sistemas orientados a objeto del nivel base o nivel de la aplicación, se descompone el comportamiento en piezas pequeñas y manejables, se crean objetos para cada pieza y se componen tales objetos en aplicaciones.

La única diferencia aquí es que la aplicación es el modelo de objetos que describe el funcionamiento de los objetos de la aplicación de usuario. Es decir, el meta-nivel es una aplicación cuyo dominio es el comportamiento de los objetos, cómo se comportan.

Al descomponer el meta-nivel en meta-componentes, se desarrolla un **modelo de ejecución de objetos relativamente genérico**. La descripción del meta-nivel se hace en función del papel que juega en la descripción del comportamiento del objeto del nivel base.

Un meta-componente o meta-objeto se encargará de cumplir cada uno de los roles que, normalmente se corresponderá con parte del comportamiento como: ejecución de objetos (tanto los mecanismos como los recursos), paso de mensajes, correspondencia de mensajes con métodos, y otros.

Cada papel a representar puede ser interpretado por varios meta-objetos diferentes y, algunas veces, un meta-objeto puede jugar varios papeles.

El comportamiento de un objeto cambia por medio de la redefinición explícita de componentes o bien, extendiendo el conjunto de roles.

2.2 La Máquina Abstracta como Conjunto de Objetos

Aunque la máquina ofrezca una funcionalidad mínima, esta incluye, típicamente, partes fundamentales del comportamiento del entorno como pueden ser la comunicación entre objetos.

Por tanto, en aras de la tan deseada flexibilidad del entorno, será necesario actuar sobre ella de forma que su comportamiento sea accesible y modificable para los objetos de usuario.

2.2.1 Elevación de los objetos de la Máquina abstracta OO

Para que el modelo de objetos y la funcionalidad implantados en la máquina abstracta pueda ser modificable para las aplicaciones, debe describirse de forma que constituya una entidad para la aplicación que quiera modificarla, es decir, **el modelo de objetos debe codificarse como un conjunto de objetos**.

Dotar a la máquina abstracta orientada a objetos de una **arquitectura reflectiva** supone que parte o toda la **funcionalidad de la máquina abstracta**, en lugar de implementarse internamente en la misma, se ofrece en un nivel superior, **como un conjunto de objetos** que componen el **modelo de la máquina**, sobre los que se puede actuar para cambiar el comportamiento de la misma.

Es decir, **el modelo de objetos debe codificarse como un conjunto de objetos**.

De esta forma, la funcionalidad pasa de estar codificada internamente en la máquina, en el lenguaje de desarrollo elegido para ello, y por tanto ser totalmente intocable, a describirse como objetos, codificados en el lenguaje definido por la propia máquina, de forma que constituya una entidad visible para aquella aplicación que quiera modificarla.

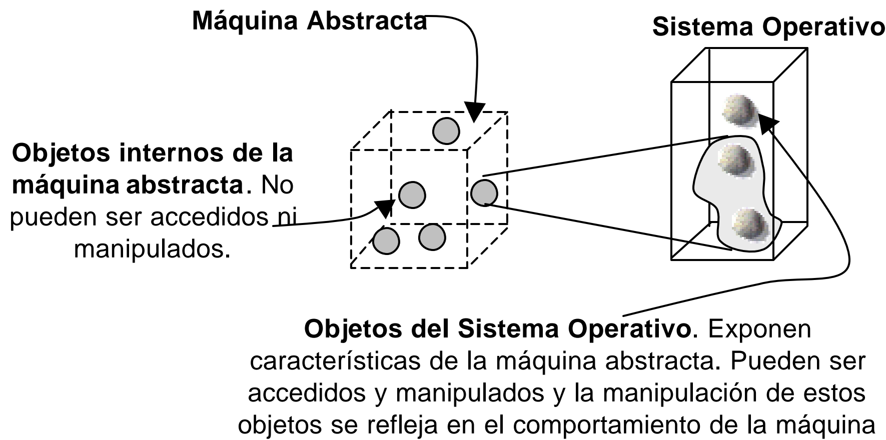


Figura 14.1 Modelo de objetos de la máquina codificado como objetos del Sistema Operativo

2.2.2 Construcción de la Máquina Abstracta Orientada a Objeto

Diseño OO de la máquina

El primer requisito para lograr ofrecer al exterior la funcionalidad de la máquina abstracta es que la construcción de esta se haga siguiendo los principios de la OO, es decir, la máquina abstracta debe diseñarse como la composición de un conjunto de objetos.

Esto se debe a que, dado que la única entidad existente es el objeto, todo aquello que no sean objetos, no podrá formar parte del mundo de objetos.

Esto implica que, toda parte de la máquina que no se describa por medio de objetos, no podrá salir del interior de la máquina, y, por tanto, no podrá ser accedida ni manipulada.

Objetos de la máquina: ¡Objetos, objetos!

Sin embargo, estos **objetos** no deben ser únicamente un elemento en tiempo de diseño o implantación, sino también **en tiempo de ejecución**.

Así, la propia máquina abstracta está formada por un conjunto de objetos que forman parte del mundo de objetos.

Además, deben ser objetos no diferenciables de los objetos de usuario. Es decir, una vez que la máquina esté compuesta por objetos, es necesario que tales objetos sean homogéneos con el resto de los objetos (no existen objetos especiales).

2.3 El Sistema Operativo como Conjunto de Objetos

La máquina abstracta OO, aunque soporta en parte la abstracción de objeto y los mecanismos que permiten comunicarlos y ejecutar sus métodos, no es suficiente para definir completamente un entorno de computación OO.

El resto de la funcionalidad necesaria la proporciona el sistema operativo a través de una serie de objetos que definen otras parcelas de la computación de objetos como la planificación [BG96], a la vez que dota al objeto de un contenido semántico completo al ofrecerle mecanismos para la ejecución controlada de sus métodos y la comunicación.

2.3.1 Adición del Sistema Operativo

La combinación reflectiva de MAOO y SOOO para construir un SIOO, no se llevó a cabo en un único paso. Se realizaron una serie de aproximaciones previas, que dieron lugar a distintos prototipos del SI, de los que se extrajeron valiosas lecciones para la construcción del prototipo final.

Máquina Abstracta Monolítica

Inicialmente, toda la funcionalidad se proporcionaba en la máquina abstracta. El SIOO carecía de SOOO y las aplicaciones solicitaban cualquier función directamente a la máquina.

La máquina abstracta se componía de un conjunto de objetos pero, únicamente, en tiempo de diseño. En tiempo de implantación, estos objetos quedaban completamente inmersos dentro de la propia máquina, eran objetos del lenguaje de programación elegido para desarrollar el prototipo.

Migración de los objetos fuera de la máquina

Una segunda aproximación supuso la migración de parte de estos objetos y por extensión, parte de la funcionalidad ofrecida por la máquina al nivel del sistema operativo.

Para ello, es necesario migrar parte de la funcionalidad ofrecida de manera interna en la máquina abstracta, a un nivel más alto y por tanto, más accesible al usuario que, se pretende que pueda instanciarlo a medida.

Sistema Operativo: Extensión de la Máquina Abstracta

Extender la máquina abstracta mediante objetos del sistema operativo (**uniformidad en torno a la OO**), que no dejan de ser objetos normales(**homogeneidad**), supone que, parte de la funcionalidad que inicialmente se encontraba implementada en la máquina, se lleve ahora a cabo a través de una serie de objetos con las mismas propiedades que cualquier otro y que conforman el entorno del sistema operativo para las aplicaciones.

Sistema Operativo: ¡Objetos, objetos¡

Al igual que sucedía con la máquina abstracta, el Sistema Operativo ofrece su funcionalidad mediante objetos que llevan a cabo aquellas tareas que, tradicionalmente, formaban parte del sistema operativo. Así, las antiguas llamadas al sistema, se traducen en este entorno por invocaciones a métodos en objetos del Sistema Operativo.

De esta forma, el **sistema operativo** se integra en el entorno de forma homogénea, ya que también ofrece su funcionalidad como un **conjunto de objetos** que extienden, modifican o complementan el comportamiento de la máquina.

3 Tipo de Reflectividad

3.1 ¿Reflectividad estructural o del comportamiento?

El objetivo de dotar a la máquina de una arquitectura reflectiva es que pueda ser extendida de manera transparente por objetos del sistema operativo, proporcionados en forma de objetos normales de usuario que se asocien a los objetos formando un espacio de ejecución para los mismos.

De esta forma, cada objeto puede adquirir propiedades adicionales (persistencia, distribución) sin necesitar modificación en los mismos. La selección del tipo de arquitectura reflectiva debe tener en cuenta este objetivo.

3.2 Reflectividad Estructural

La reflectividad estructural ya existe en cierta medida, en el modelo de objetos de la máquina desde las primeras versiones de la misma. De hecho, la MAOO se diseñó aplicando los principios de OO, lo que da como resultado una máquina abstracta en la que todo son objetos, los métodos, las instrucciones, las excepciones, etc. Este principio de diseño, favorece la reflectividad estructural, al facilitar la exposición (*reification*) al exterior la estructura del sistema.

En las primeras implantaciones[Izq96], la meta-interfaz no permitía más que consultar información acerca de la estructura de los objetos. Se trataba además de una reflectividad estática que permitía que un objeto consultase su estructura interna (su representación en tiempo de ejecución), pero no permitía que la modificase. Ejemplo de ello son métodos como getClass(), que permiten interrogar a la máquina acerca de la clase de un objeto.

Sin embargo, dado que la propia máquina abstracta se diseñó desde un principio aplicando los más rigurosos principios de OO [Izq96], resultó relativamente sencillo extender esta reflectividad estructural haciéndola más completa. Así, se extiende la meta-interfaz para que abarque más aspectos del modelo de objetos, como por ejemplo, solicitar la jerarquía de un objeto, y se implanta una reflectividad estructural dinámica que permite crear y modificar clases en tiempo de ejecución, etc.

Un tratamiento más detallado se verá en el Capítulo XV y en el Anexo C.

3.3 Reflectividad del Comportamiento

Para lograr el objetivo de extender la máquina adaptando su comportamiento a las aplicaciones que se ejecutan sobre ella, es la reflectividad de comportamiento el campo que mejor lo permite.

Dado que la reflectividad encaja especialmente bien con los conceptos de objeto y OO, una extensión natural de la exposición de la estructura, es organizar el control del

comportamiento de un sistema de computación OO (su meta-interfaz) a través de un conjunto de objetos denominados meta-objetos[BG96].

Se trata de exponer el comportamiento de la máquina – su forma de ejecutar las aplicaciones –, para lo que es necesario cosificar los elementos de la máquina que hacen que funcionen los objetos del nivel base.

Los meta-objetos pueden representar varias características del entorno de ejecución, como la representación de la ejecución, la ejecución en sí, la comunicación o la localización, de tal forma que, la especialización de estos meta-objetos puede extender y modificar el contexto de ejecución de un conjunto específico de objetos del nivel base.

La reflectividad del comportamiento también puede expresar y controlar la gestión de recursos, no a nivel de un objeto individual, sino a un nivel mucho más amplio como la planificación, el agrupamiento de objetos, etc.

Esto ayuda a ejercer un control de grano mucho más fino, por ejemplo para planificación y balance de carga, en oposición a algún algoritmo global y fijo que, normalmente, se optimiza para alguna clase de aplicación de la media.

3.3.1 Utilización del Paradigma de OO para describir el MetaNivel

Para introducir la reflectividad en el sistema de manera uniforme, lo más interesante es que el modelo del meta-sistema esté descrito utilizando el mismo paradigma OO que el propio sistema. De este modo, tanto el modelo, como las operaciones de exposición y reflexión pueden ser utilizados de manera uniforme dentro del mismo paradigma.

Así, el **nivel base** estará compuesto por los objetos que definen la aplicación (lo que hace el objeto), mientras el **meta-nivel** estará compuesto por otros objetos (**uniformidad**) que definen el entorno de ejecución de los objetos del nivel base o de aplicación.

3.3.2 Los Meta-Objetos dan Soporte a los Objetos Base

Cada objeto del nivel base tendrá un **meta-nivel conceptual** o **meta-espacio** asociado. Este meta-espacio estará formado por un conjunto de **meta-objetos**.

Descripción Separada del Nivel Base y el Meta-Nivel

En esta tesis, al igual que en otros trabajos como Apertos [Yok92], se propone describir separadamente el objeto y su meta-nivel. De esta forma, los programadores pueden centrar su atención en describir los métodos que solucionan los problemas. Otros problemas, como encontrar el objeto destino o gestionar el envío de mensajes (o invocación de métodos), se describen en el meta-nivel utilizando el mismo marco OO.

Así, por ejemplo, un objeto del nivel base, puede describirse independientemente de su ámbito de actuación y gestión de recursos (local o remoto). Este aspecto de su ejecución, si puede solicitar recursos remotos o únicamente recursos locales y otros, se describe en el meta-nivel.

El Meta-Espacio

Cada uno de los meta-objetos que componen el meta-espacio asociado al objeto base, describe algún aspecto del comportamiento del mismo, supervisando la ejecución de los objetos de este nivel, de forma similar a como se hace en CodA[McA95],

Apertos[Yok92] o en la ampliación del modelo de objetos de Java descrita en [GK97b] que pretende dotarla de un Modelo de Objetos Activo.

Para hacer posible esta supervisión, es necesario exponer algunos aspectos de la computación del nivel base, es decir, es necesario hacer explícitos algunos eventos que, en la implantación no reflectiva formaban parte de la implantación interna de la máquina abstracta.

Este conjunto de meta-objetos asociados a un objeto, que le sirven de soporte y le permiten especializar el comportamiento de la máquina colaborando con ella en la ejecución de algunas instrucciones, será el que conforme el entorno de ejecución óptimo para el objeto y con ello, determinarán sin ambigüedad el comportamiento particular del mismo respecto a cómo y cuándo lleva a cabo la ejecución de sus métodos.

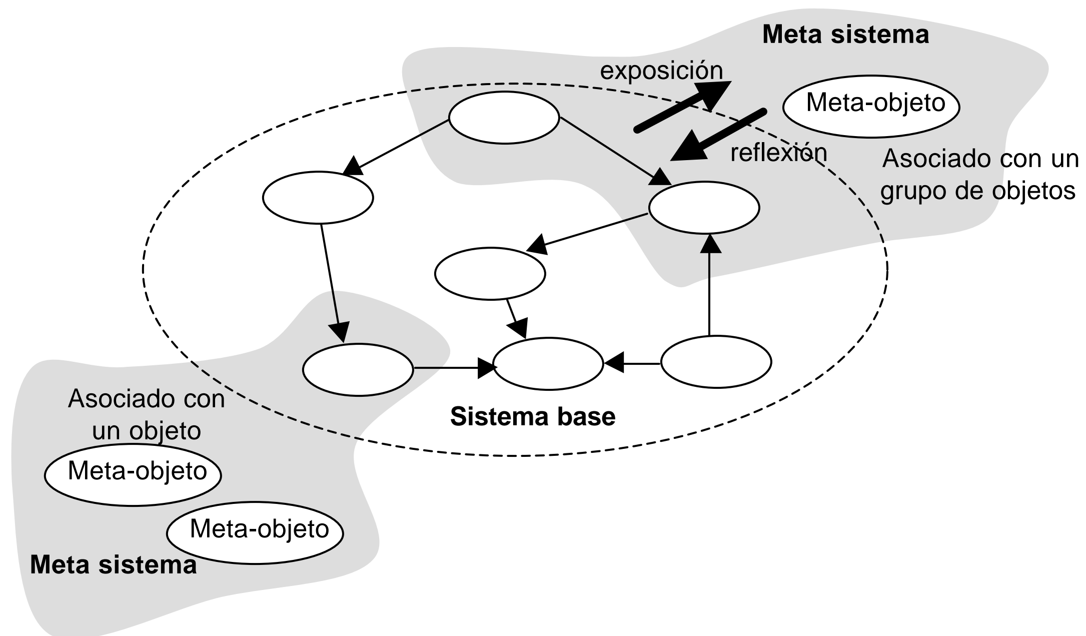


Figura 14.2 Modelo Computacional de la reflectividad del comportamiento

Meta-Interfaz

De esta forma, se permite que los objetos del nivel base consulten y manipulen al meta-sistema. La interfaz con el meta-nivel permite ajustar los servicios proporcionados por una API del nivel base a las necesidades específicas de las aplicaciones en tiempo de ejecución dotando al sistema de características de flexibilidad y adaptabilidad.

Esta aproximación reflectiva transforma el sistema de :

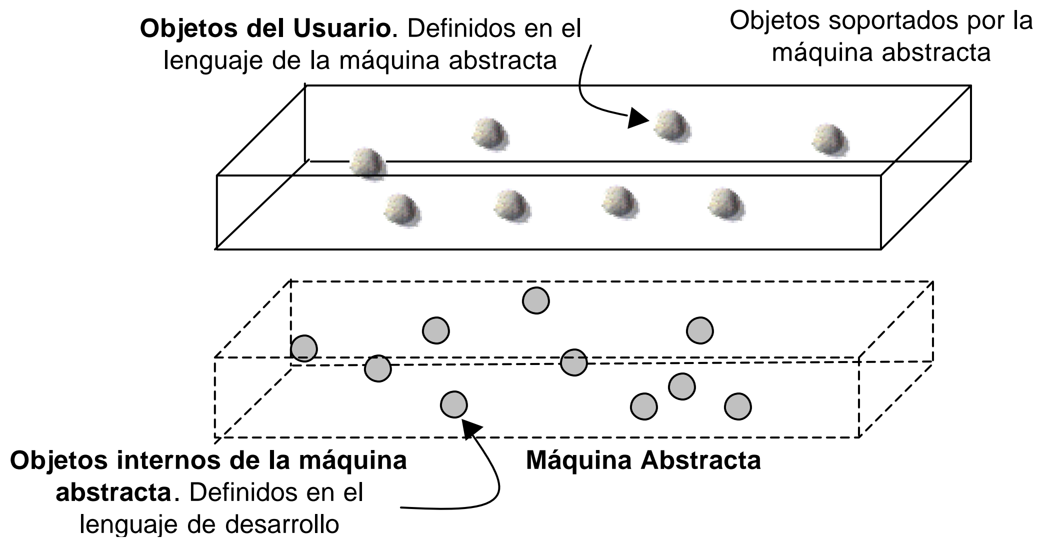


Figura 14.3 Objetos de la máquina y objetos de usuario. Dos niveles de objetos distintos

En:

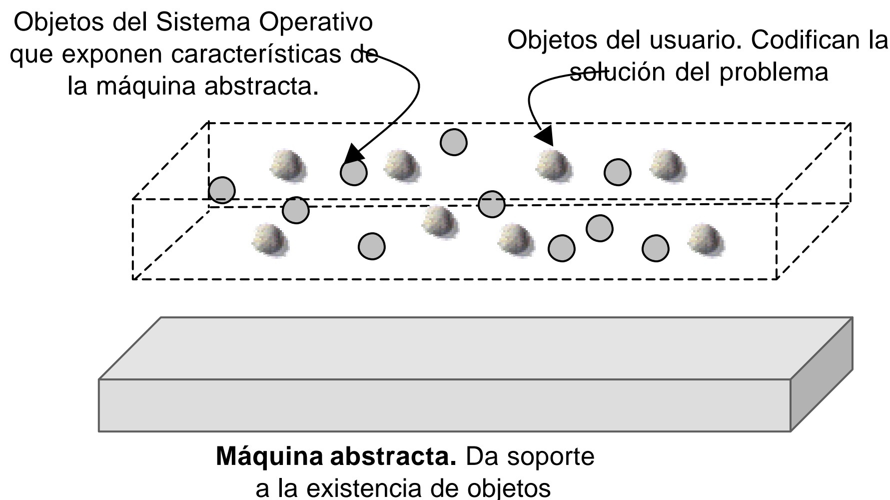


Figura 14.4 Objetos de la máquina y objetos de usuario con soporte similar

4 Arquitectura Reflectiva Propuesta

La propuesta que se hace en esta tesis de organizar la máquina abstracta y el sistema operativo utilizando el mecanismo de reflectividad para ello, plantea un conjunto de preguntas, cuya respuesta definirá completamente la arquitectura reflectiva.

4.1 Torre Reflectiva: Dos Niveles de objetos

Un sistema reflectivo OO, se organiza en dos o más niveles, que consituyen la torre reflectiva. Algunos sistemas, como Apertos[Yok92], tienen varios niveles de

reflectividad lo que los convierte en sistemas complejos, difíciles de entender para usuarios poco avezados y con un bajo rendimiento.

La propuesta de esta tesis es adoptar una torre con únicamente, **dos niveles**. Un **nivel base**, los objetos de la aplicación, y un **meta-nivel**, los objetos del sistema operativo que extienden y complementan el funcionamiento de la máquina abstracta, describiendo el comportamiento de los objetos del nivel base. Gracias a esto, se consigue exponer los aspectos que nos interesan de forma más sencilla y eficiente.

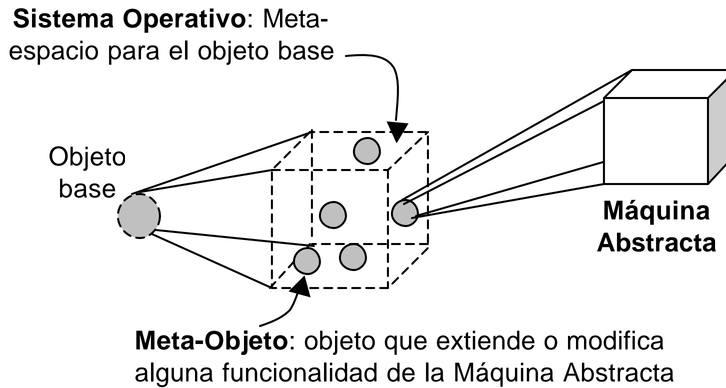


Figura 14.5 Dos niveles de Objetos: El Nivel Base y el Sistema Operativo

4.2 Modelo de Reflectividad

Al diseñar un sistema reflectivo, una de las primeras preguntas que hay que responder es cómo se representa la reflectividad, o dicho de otra forma, cuál es el modelo de reflectividad elegido.

4.2.1 Elección del Modelo

En el Capítulo XII se presentaron detalladamente cada uno de ellos. Sirva aquí con señalar cuáles fueron las razones que nos inducen a elegir uno y descartar los demás.

- En primer lugar, el **Modelo de MetaClase** adolecía de parcialidad en la flexibilidad. Al asociar meta-objeto con clase, todos los objetos de esa clase se veían obligados a utilizar la misma semántica, o bien a implantar complicados mecanismos como en MetaJava[GK97a].
- En el **Modelo Meta-Mensaje**, no es posible representar el comportamiento de los objetos y no permite guardar información aunque al ser el paso de mensajes la herramienta más potente para comenzar y controlar el flujo de ejecución, será un buen candidato para combinarlo con el siguiente.
- El **Modelo Meta-Objeto** es, con mucho el más extendido y estudiado. Permite que cada objeto personalice su entorno de ejecución, definiendo su comportamiento en el sistema, a la vez que es posible registrar información importante en la vida de ese objeto. Todo ello, nos inclina a adoptar este último como propuesta en esta tesis.

4.2.2 El Modelo de Meta-Objetos

La propuesta que se hace en esta tesis para el meta-nivel supone representar la funcionalidad a exponer mediante objetos. O lo que es lo mismo, se representan las características estructurales y las características de ejecución de las aplicaciones en forma de objetos (**meta-objetos**) que componen el entorno computacional por defecto.

El uso de meta-objetos, además de permitir un poder de representación del entorno en tiempo de ejecución mayor que en otros casos, tiene varias características que lo hacen atractivo.

Uniformidad

En primer lugar, el meta-objeto no rompe la uniformidad del sistema en torno al concepto único de objeto y el entorno en tiempo de ejecución se representa en forma de objetos con los que se trabaja mediante el uso de referencias y el paso de mensajes, como con cualquier otro objeto.

Flexibilidad

La especialización de los meta-objetos es posible gracias a que estos están asociados a objetos lo que hará posible la adaptación del entorno de ejecución de las aplicaciones ofreciendo a las mismas la posibilidad de cambiar la forma de representar los datos, estrategias de ejecución, mecanismos y protocolos [BG96].

Este entorno de computación flexible proporciona una representación de las características relevantes e importantes del sistema en términos del sistema mismo, al tiempo que evita el problema de parcialidad del que adolecía el modelo de Meta-Clase.

4.3 El Nivel Base

Los objetos del nivel base, que heredan todos en último término de la clase **OBJETO** implantada en la MAOO, están formados por datos y métodos que modelan una entidad del problema y que interaccionan entre sí gracias a la invocación de métodos, lo que se traduce por paso de mensajes entre objetos. Este es el único mecanismo ofrecido para comunicar objetos y proporciona además los principios básicos para la sincronización de los mismos.

4.3.1 Semántica de los objetos del Nivel Base

En el modelo de computación propuesto en esta tesis, un sistema está formado por un conjunto de objetos y cada objeto se puede entender como un **mecanismo autónomo de procesamiento**. Esto implica que los objetos deben estar capacitados para gestionar su propia computación de forma flexible, adaptable y, fundamentalmente, homogénea. De este modo, el objeto se convierte en un **servidor de sus métodos**.

Sin embargo, la máquina abstracta ofrece un soporte muy simple a los objetos, como se verá en el Capítulo XV. Simplemente ofrece el objeto como entidad autónoma de procesamiento, sin ofrecer ningún mecanismo de gestión de la ejecución de los métodos.

4.4 El Meta-Nivel

Como se ha visto en el apartado anterior, el objeto base, tal como está definido, no está dotado de los mecanismos necesarios que le permitan ofrecer la semántica que se le exige.

Por tanto, para poder ofrecer este comportamiento, es necesario complementar el objeto base con un metaespacio que dota al mismo de las capacidades necesarias para convertirse en un ente autónomo.

La propuesta que se hace en esta tesis para el meta-nivel, supone exponer estos aspectos de la máquina mediante un conjunto de objetos proporcionados en el nivel inmediatamente superior, que componen el meta-nivel, y especifica que cada objeto del nivel base tiene asociado un meta-nivel conceptual.

Es decir, se trata de asociar a cada objeto un meta-espacio, compuesto por uno o varios meta-objetos que expongan distintos aspectos del entorno en tiempo de ejecución del objeto base. Esta asociación es dinámica en tiempo de ejecución, puede eliminarse o cambiarse en cualquier momento y permite cambiar la semántica del objeto dinámicamente, lo que confiere gran flexibilidad al entorno.

4.4.1 Aspectos de la máquina que migran al meta-nivel

Recepción de Mensajes

Cuando se invoca un método en un objeto – o se le envía mensaje– este debe ser capaz de recibirlo aunque en este momento no sea capaz de procesarlo.

Para llevar a cabo la recepción de mensajes, el objeto debe tener la habilidad de:

- **Conocer y Acceder a su estado interno en tiempo de ejecución**. El objeto debe tener acceso a información que representa su estado de ejecución. Esta información, que formará parte del meta-nivel del objeto, será gestionada siguiendo una política variable en función del estado y la finalidad del objeto.
- **Retrasar su ejecución**. En función de lo que dicte la política anterior, el objeto puede decidir retrasar el comienzo de la ejecución del método solicitado, de acuerdo a su estado interno o a cualesquiera circunstancias que influyan en su ejecución.

  Para ello, la máquina subyacente debe ofrecer un mecanismo que permita retrasar la ejecución de un método y, el objeto debe contar, entre los meta-datos que describen su estado en tiempo de ejecución con una **lista de métodos pendientes** donde poder registrar los retrasos que se dispongan.

Esta habilidad de decidir acerca de la ejecución de sus métodos en función de datos del meta-nivel, formará parte del meta-espacio del objeto base.

Procesamiento del Mensaje

Si se recibe un mensaje y este resulta aceptable, el objeto puede analizar su estado y determinar que su ejecución comience. La ejecución del método es una combinación de acciones que suponen acceder a su estado interno y actualizarlo, si procede, enviar mensajes a otros objeto incluido a sí mismo, crear nuevos objetos, etc.

Dado que el objeto es la unidad de concurrencia, debe ser capaz de llevar a cabo una gestión completa de la concurrencia permitida, con la posibilidad de actuar de distintas formas en caso de no ser adecuada la ejecución de un método.

Así, durante la ejecución de un método, un objeto puede, basándose en ciertos factores de control de la concurrencia, determinar suspender la ejecución de un método. Incluso puede determinar no comenzar su ejecución, aunque el mensaje sea aceptable.

En cualquier caso, el objeto puede, en función de su estado interno y de otros factores que influyen en la ejecución de sus métodos, realizar una planificación sobre la ejecución de sus métodos, retrasando su ejecución o reanudando los métodos pendientes.

Tanto la política de control de la concurrencia, como la política de planificación del objeto, dependen de la instancia concreta del objeto y se llevan a cabo en el meta-nivel, accediendo a los meta-datos del objeto para determinar la conveniencia o no de la ejecución o reanudación de un método.

Transmisión de Respuestas/Excepciones

Es imprescindible que cada objeto sea capaz de enviar respuestas al objeto que le había solicitado el servicio. Aunque no es necesario que estas respuestas sean los resultados de la ejecución correcta y completa del método solicitado, sino que puede suceder que se transmita la excepción resultante de algún problema en la ejecución.

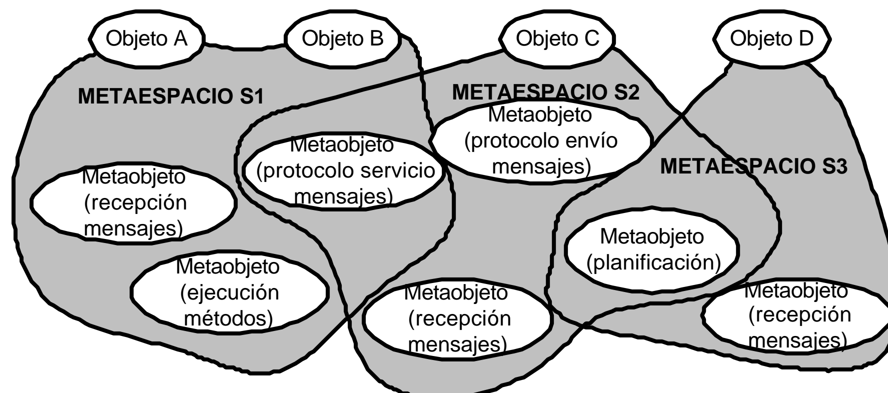


Figura 14.6 Arquitectura propuesta. El MetaNivel da soporte al nivel base

4.4.2 El Meta Objeto

Al igual que los objetos base, los metaobjetos también son objetos con datos y métodos que se comunican igualmente mediante la invocación de métodos en otros objetos aunque, en lugar de derivar de la clase **OBJETO**, derivan de la clase **METAOBJETO**, también soportada por la máquina abstracta.

Sin embargo, son dos las diferencias fundamentales respecto a los objetos del nivel base.

Acceso a la Información

Cada meta-objeto tendrá los datos y métodos necesarios para llevar a cabo su tarea, y la información necesaria o acceso a la información necesaria en tiempo de ejecución del objeto base al que esté asociado(métodos en ejecución, métodos solicitados, etc.).

Por tanto, **los datos para gestión del objeto son a su vez objetos en el metanivel**[Yok92].

De esta forma, se pueden implementar mecanismos de gestión del comportamiento del objeto, que tengan acceso a estos meta-datos:

- mecanismos que permiten inspeccionar el estado interno del objeto en tiempo de ejecución,
- cambiar la política de planificación del objeto,
- enviar mensajes, etc.

Ejecución de Métodos

Por otro lado, para evitar una sobrecarga excesiva en la sincronización, la ejecución de los métodos en los objetos del meta-nivel, se lleva a cabo de forma síncrona con la ejecución de los métodos del objeto del nivel base que los invoca o provoca su ejecución.

Es decir, la computación en el nivel base se suspende mientras el meta-objeto procesa el evento.

De esta forma, además, el meta-objeto tiene control absoluto sobre las actividades que se llevan a cabo en el el objeto del nivel base.

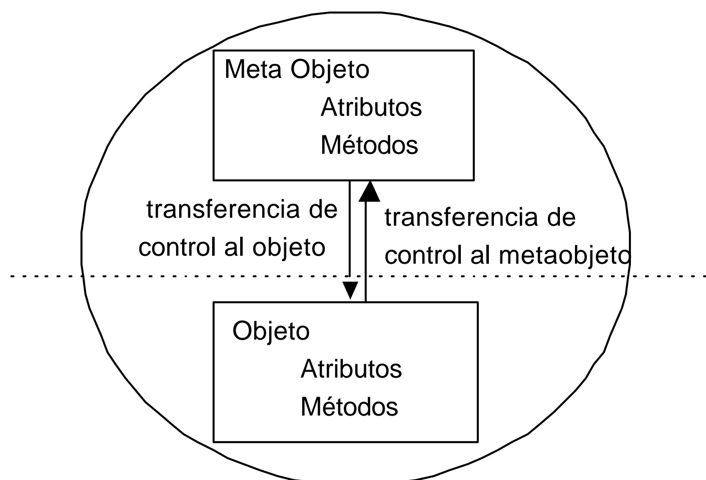


Figura 14.7 Objetos: El Nivel Base y el Meta-Objeto

4.4.3 Representación del objeto base en tiempo de ejecución

Para trabajar con meta-objetos, son necesarios algunos cambios en el kernel de la máquina[Fer89].

Soporte en el entorno de ejecución base al enlace base-meta

El primero de ellos, consiste en modificar la representación que la máquina soporta de los objetos en tiempo de ejecución, modificando la definición de la clase **OBJECT**, para establecer, a partir de ella, la asociación entre el nivel-base y el meta-nivel. A través de ese enlace, el nivel base puede comunicarse con el meta-nivel.

De esta forma es posible cambiar el meta-objeto simplemente cambiando el valor asociado a esa posición del objeto.

Soporte en el entorno a la comunicación Base-Meta

El segundo cambio se refiere a la definición de la función primitiva de paso de mensajes, que debe chequear si hay un meta-objeto asociado al objeto, en cuyo caso, las acciones de envío del mensaje, procesamiento del mismo, ejecución del método, y otras, podrían delegarse en uno de los meta-objetos.

Debe considerar también la posibilidad de que el destinatario del mensaje sea un meta-objeto (reflectividad explícita), en cuyo caso, no se delegaría, para evitar la regresión infinita.

4.5 Transferencia de Control al Meta-Nivel

La comunicación entre el nivel base y el meta-nivel tiene como objetivo pasar el control del primero al segundo para llevar a cabo la meta-computación y, posteriormente, devolverlo al nivel base para terminar de ejecutar la computación en este nivel.

En cuanto a la transferencia de control, es decir, al momento, causas y consecuencias que tendrá la transferencia de control del nivel-base al meta-nivel, se articula en torno a tres puntos: **Mecanismo Uniforme**, **Reflectividad Explícita** e **Implícita**.

Una vez que la meta-entidad tiene el control y realiza completamente la meta-computación correspondiente, el flujo de computación vuelve al nivel base para que la entidad base complete su ejecución. En este caso, el flujo de control vuelve del meta-nivel al nivel base.

4.5.1 Reflectividad Implícita: Invocación de Métodos ® Meta-Computación

Es una de las formas de activar la meta-computación que se implantan en la arquitectura propuesta.

La computación reflectiva se activará de forma automática en determinados momentos del proceso de ejecución de un objeto. Es decir, en el proceso normal de ejecución de objetos por parte de la máquina abstracta subyacente, existen tareas que ésta no puede llevar a cabo, es decir, que no están codificadas internamente en el lenguaje de implantación de la misma. Debido a esto, es necesario definir estas tareas en el meta-nivel para completar el proceso de ejecución.

La ejecución de algunas acciones en el nivel base, como la invocación de métodos, la finalización de la ejecución de un método y otras que se describirán en el Capítulo XV, provoca que la propia máquina abstracta transmita el control al meta-nivel

continuando la ejecución en el meta-nivel, de forma semejante a un sistema de eventos[Rei97].

Cada vez que se produzca alguna de dichas acciones, la máquina abstracta lo comunicará al meta-sistema, es decir, transferirá el control al meta-nivel, enviando un mensaje al objeto que implemente tal característica en el meta-nivel y, una vez terminado, retornará el control al nivel base para proseguir su ejecución.

El meta-objeto concreto que recibirá el control, dependerá de la acción que haya provocado la ruptura de control. En cualquier caso evaluará la situación y reaccionará de una forma específica, que podrá variar en función de la especialización que cada objeto o conjunto de objetos relacionados haya hecho del meta-objeto que gestiona ese aspecto de la computación.

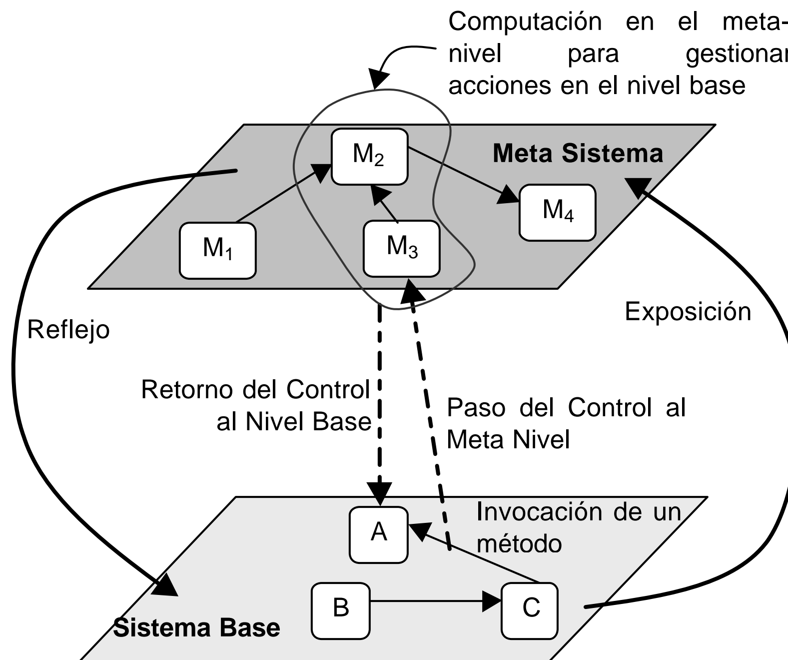


Figura 14.8 Reflectividad Implícita: Paso de Control ante determinados eventos en el nivel base

La correspondencia entre las acciones del nivel base que provocan la transferencia y la meta-computación que provocan es dinámica, es decir, en tiempo de ejecución es posible cambiar el meta-objeto que gestiona ese aspecto de la computación del nivel base.

En la propuesta que se hace en esta tesis, las tareas del nivel base que provocan la transferencia de control al meta-nivel son la ejecución de métodos, la planificación de tareas para la ejecución y, fundamentalmente la invocación de métodos o paso de mensajes, fuente principal de transferencia de control del nivel base al meta-nivel.

4.5.2 Reflectividad Explícita

Por otro lado, el sistema reflectivo propuesto en esta tesis permite la reflectividad explícita y promueve su utilización como medio para adaptar el entorno del objeto en tiempo de ejecución.

Cuando la aplicación lo necesite, también puede invocar directamente métodos en cualquier objeto del meta-nivel del que disponga de identificador y permisos, igual que para cualquier otro objeto del mundo de objetos.

Cada objeto estará dotado de métodos para la introspección, necesarios para obtener las referencias de sus meta-objetos en tiempo de ejecución, lo que supone acceder a información de soporte en tiempo de ejecución, Reflectividad estructural.

Por ejemplo, se puede invocar un método en el meta-objeto que envía mensajes con el fin de registrar el envío y recepción de mensajes y llevar a cabo acciones de depurado.

En el Capítulo XV y en el Anexo C se ofrece una visión más detallada de cómo se implementa la reflectividad estructural y la interfaz que ofrece.

4.5.3 Transferencia de Control Uniforme

Para implantar la reflectividad explícita, es necesario ofrecer a los objetos de usuario, un mecanismo de comunicación que le permita invocar métodos de los objetos del meta-nivel.

Debido a que la representación de las características expuestas se lleva a cabo utilizando la abstracción de objetos, se propone utilizar el mismo mecanismo de comunicación para comunicar objetos que para transmitir el control entre el nivel base y el meta-nivel, dado que, al fin y al cabo, también se trata de comunicar objetos.

Este mecanismo no puede ser otro que el paso de mensajes. Sin embargo, esto supone modificaciones referidas a la definición de la instrucción primitiva de invocación a métodos.

Esta instrucción debe pasar el control al meta-espacio y comenzar la meta-computación, a menos que se trate de una invocación a un método de un meta-objeto (Véase en los capítulos XV y XVI, los apartados referidos a la Comunicación entre Objetos).

A partir de este momento, es posible que los meta-objetos colaboren, mediante la invocación de métodos unos a otros para implementar el entorno del objeto base.

Solución como esta – hacer explícitos los meta-objetos que, de esta forma, son accedidos vía el paso de mensajes normal – puede encontrarse también en CLOS.

4.5.4 Una Arquitectura Resultado de la composición de objetos

Construir un entorno de computación OO donde el comportamiento esté definido por objetos a nivel usuario accesibles vía el mecanismo de invocación a métodos tradicional, además de ejecutar sus métodos por la transferencia de control implícita por parte de la máquina abstracta, supone ofrecer al usuario una arquitectura con múltiples ventajas.

Flexibilidad

Añade flexibilidad al entorno, que, tradicionalmente era fijo, permitiendo cambiar los objetos que componen el entorno de soporte por otros, que den soporte de otra forma a las mismas características.

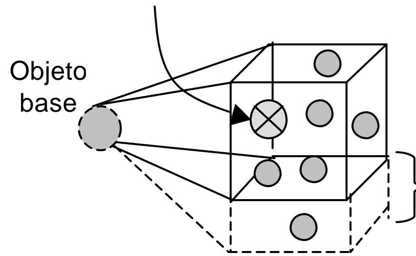


Figura 14.9 Modificación del meta-espacio: Entorno Flexible

Uniformidad

Extiende la máquina de forma transparente gracias a la incorporación de nuevos objetos ofrecidos por el sistema operativo. Se mantiene la uniformidad, ya que los objetos del sistema operativo son objetos normales de usuario.

Se ve por tanto que para extender la máquina, la reflectividad es la técnica más prometedora. Para ello, habrá de identificar aquellos elementos de la máquina que ofrecen el funcionamiento básico a los objetos del nivel base y exponerlos. De esta forma se pueden modificar, cambiar uno por otros, etc.

5 Objetivos del Meta-Nivel: Selección de Meta-Objetos

Para terminar de definir la arquitectura reflectiva propuesta, se describen a continuación qué aspectos de la máquina se exponen, lo que se traduce en cuál es el MOP que se ofrece a más alto nivel.

Esto determinará el conjunto de meta-objetos por defecto que compondrán el meta-espacio de los objetos del nivel base.

Los trabajos que en este sentido se están realizando, ofrecen diversas alternativas.

- **Meta-Arquitectura de Grano Fino**. Especifica una meta-arquitectura compuesta de un conjunto de meta-componentes que exponen diversos aspectos del comportamiento del objeto a nivel base. CodA[McA95a] es un ejemplo de esta línea y define siete componentes por defecto presentes en el meta-nivel de todos los objetos: **Send**, **Accept**, **Queue**, **Receive**, **Protocol**, **Execution** y **State**. Si bien permite un mayor nivel de especificación, en algunos casos, la presencia de tantos meta-objetos puede suponer un aumento de la complejidad a la hora de "*juntar todas las piezas*" para construir un entorno de ejecución.
- **Meta-Arquitectura de Grano Medio**. Lo habitual es exponer, en una primera aproximación, la comunicación entre objetos o paso de mensajes como elemento clave de la arquitectura reflectiva. Se puede señalar a Merlin[AK95] como claro exponente.
- **Exposición de aspectos de implantación tradicionales de los Sistemas Operativos**. Expone los aspectos del comportamiento del objeto a nivel del Sistema Operativo como pueden ser la gestión de memoria, gestión del almacenamiento,

faltas de página, gestores de dispositivo o planificación. El ejemplo más significativo de esta corriente es Apertos[Yok92].

En este apartado se presentan los distintos aspectos de la máquina que se exponen, así como las razones que nos llevaron a exponer tales aspectos y los meta-objetos que los describen a nivel de objetos. No pretende ser una exposición exhaustiva de la estructura, funcionalidad y comportamiento de los meta-objetos, sino una breve introducción a los mismos.

Un tratamiento en profundidad, acompañado de una descripción más detallada de los distintos aspectos y los meta-objetos, se deja para los dos capítulos posteriores, en particular para el capítulo XVI.

5.1 Objetivos del Meta-Nivel

Los objetivos del meta-nivel referidos al entorno de computación de los objetos, se refieren, fundamentalmente, a los mecanismos que es necesario proporcionar para dotar al sistema de un modelo de objetos activo.

Otros objetivos, como los referidos con la persistencia en el sistema, se salen del ámbito de esta tesis y ya han sido tratados en [Alv97].

Como base de la arquitectura reflectiva, proponemos en esta tesis tres aspectos de la ejecución de objetos que darán lugar a un conjunto de meta-objetos. Este meta-espacio es común a todos los objetos y, aunque no cubre todos los posibles aspectos del comportamiento de los objetos, sí expone los fundamentales desde el punto de vista del entorno de ejecución y cubre los aspectos esenciales del modelo de objetos en lo que se refiere a la concurrencia en la ejecución de métodos.

Comunicación entre Objetos o Invocación de Métodos

No sólo se exponen las características del envío y recepción de mensajes, sino también la política de aceptación de los mismos en el lado del receptor, incluyendo la posibilidad de asignar prioridades a los mensajes.

Control de la Concurrencia dentro de los Objetos o Sincronización Intra-Objetos

Dado que los objetos se definen como agentes autónomos de ejecución con capacidad para ejecutar múltiples métodos simultáneamente, es necesario establecer, en el borde del objeto, la política de sincronización que determine en qué orden y bajo qué condiciones se ejecutarán los métodos.

Ejecución Concurrente de Métodos o Sincronización Entre-Objetos

En un entorno concurrente, permite especificar una política concreta de planificación de ejecución de los distintos métodos, sean estos de un mismo o distintos objetos.

5.2 Meta-Objetos Mensajero

El paso de mensajes es una de las nociones fundamentales sobre la que se basan los sistemas orientados a objetos por lo que redefinir su semántica es una herramienta muy poderosa.

Lo habitual en aquellos sistemas reflectivos que exponen la comunicación entre objetos, es exponer dos aspectos del tratamiento de los mensajes: el proceso de búsqueda y ejecución del método adecuado para el mensaje recibido. Sin embargo, otros sistemas como CodA[McA93], permiten una redefinición mucho más amplia.

Ambos aspectos son actividades relacionadas con la tarea del servidor para servir el mensaje recibido.

Para incrementar la flexibilidad, también se expone el envío del mensaje en el lado del cliente. De esta forma, un objeto puede definir su entorno de comunicación (local, remoto, ...).

5.3 Meta-Objeto Sincronizador

Un modo de introducir el control de la concurrencia en el sistema es separar la descripción de la sincronización del código secuencial de tal modo que el código de sincronización no se mezcle con el código de los métodos y se consiga una separación más clara de algoritmo y sincronización.

La idea básica del mecanismo de sincronización compartido con [Rei97, CV98, VA98] es dividir el estado del objeto concurrente en parte secuencial y parte concurrente como se muestra en la Figura 14.10.

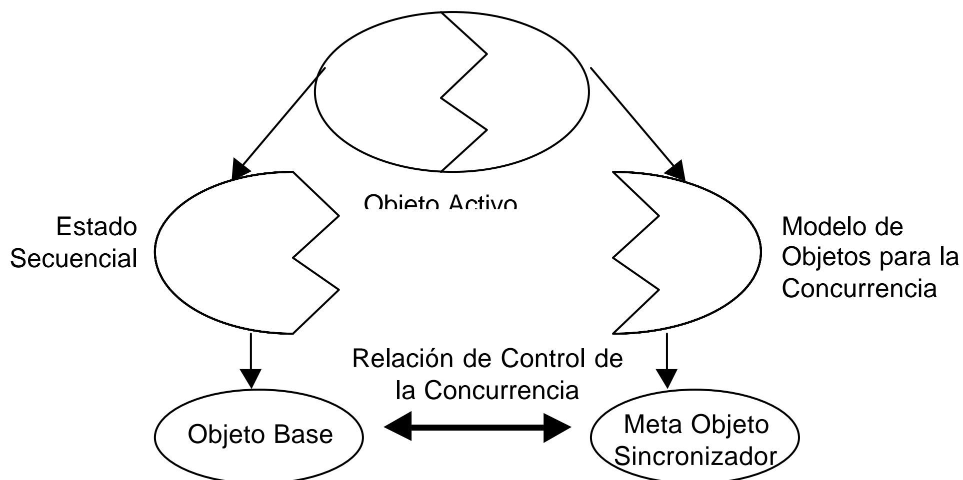


Figura 14.10 Implantación objetos activos. Control de la Concurrencia en el Meta-espacio

La parte secuencial contiene el estado y los métodos que implementan el comportamiento del objeto secuencial. Este es el objeto base. Por su parte, el objeto concurrente contiene el estado y los métodos que implementan el comportamiento del objeto de cara a sincronizar la ejecución concurrente de varios métodos del objeto base. Este es el meta-objeto que hemos dado en llamar **Sincronizador**.

5.4 Meta-Objeto Planificador

Una de las características funcionales más importantes es dotar a los objetos de concurrencia interna. Para ello, los objetos se definen como entidades concurrentes multi-hilo, capaces de servir varios métodos concurrentemente.

Otro requisito funcional relevante en un entorno de ejecución concurrente es permitir la concurrencia externa, es decir, la posibilidad de que varios objetos compitan por los recursos necesarios para ejecutar sus métodos, en concreto, por el control de la máquina subyacente.

De todo esto se deriva la necesidad de establecer una política de planificación que determine el orden de ejecución de las tareas tanto internas como externas.

5.5 Modelo de objetos implantado en el Meta-Espacio

Asociando con un objeto o conjunto de objetos base meta-objetos como los que aquí se mencionan, se implementa en el meta-nivel el modelo de objetos activo transformando un objeto pasivo en una entidad similar a los actores[AK98], de forma semejante a lo que se propone en[GK97b].

Los objetos activos son una extensión del modelo pasivo donde el objeto pasivo implementa la parte funcional del objeto activo y el comportamiento activo se implementa en el meta-nivel.

5.6 Funcionamiento Global del nivel-base y el meta-nivel

En la propuesta actual, el principal evento que provoca la transferencia de control del nivel base al meta-nivel es la invocación de un método en un objeto. Se transfiere el control al meta-objeto mensajero que envía el mensaje al objeto destinatario.

La máquina abstracta, que implementa la invocación a métodos a bajo nivel, transferirá el control al meta-nivel del lado del emisor, cuando encuentre la instrucción de paso de mensajes.

Cuando se trate del lado del receptor, también se transfiere el control al meta-objeto mensajero que se encarga de recibir los mensajes.

La recepción de un mensaje supone la transferencia de control al meta-objeto Sincronizador para la posible aceptación de mensajes. Este meta-objeto también ve invocados sus métodos cuando un método finaliza o cuando se para por cualquier motivo (E/S, sincronización explícita con semáforos o similares, ...).

Por último, la aceptación de un mensaje para servirlo, supone la aplicación de una política de planificación, de acuerdo a la prioridad del método solicitado, el remitente o algún otro criterio.

Por tanto, la aceptación de un mensaje supone la activación del meta-objeto planificador para que le asigne un lugar en el mundo de actividades del objeto base.

Esto se verá nuevamente, en más profundidad en el Capítulo XVI.

Capítulo XV

La Máquina Abstracta. Entorno de Computación Básico

A pesar de que en esta tesis se propone estructurar el entorno OO como la combinación reflectiva de Sistema Operativo y Máquina Abstracta, existen algunas tareas básicas de soporte a la abstracción de objeto que la máquina lleva a cabo por sí misma.

Entre ellas, se encuentran el soporte a la existencia de objetos, derivado del propio diseño de la máquina, que conoce y gestiona el concepto de objeto como unidad básica de computación. Este soporte a la existencia incluye la identificación, comunicación y ejecución de objetos.

Además de este soporte básico, la máquina abstracta incluye el soporte de bajo nivel a los componentes de seguridad, distribución, concurrencia y persistencia, que se especifican en más profundidad a nivel del Sistema Operativo.

Igualmente, la máquina abstracta debe estar diseñada y construida con una estructura reflectiva que le permita pasar el control de la ejecución del nivel base al meta-nivel, cuando el contexto de ejecución del objeto así lo requiera.

1 Arquitectura de Referencia de una Máquina Abstracta

A continuación, se describe una arquitectura de referencia para la máquina abstracta orientada a objetos con las propiedades necesarias para dar soporte a un sistema integral orientado a objetos.

A partir de esta arquitectura de referencia, se dispone de varias implantaciones – una de las cuales se ofrece en el Capítulo XVII – que se desarrollaron, y se continúan desarrollando, y que han supuesto una valiosísima plataforma para experimentar y apoyar lo que en esta tesis se propone.

1.1 Propiedades fundamentales de una máquina abstracta para un SIOO

Las características fundamentales de la máquina abstracta se derivan directamente de los objetivos fundamentales ya vistos en el Capítulo V para un SIOO y en los capítulos VIII, IX y X para un Sistema de Computación OO.

Modelo único de objetos implementado por la máquina

La máquina abstracta que se utilice deberá proporcionar las siguientes características del modelo único de objetos:

- Identidad única de objetos
- Clases y relaciones de herencia múltiple (es-un), agregación (es-parte-de) y relaciones generales de asociación (asociado-con).
- Comprobación de tipos, fundamentalmente en tiempo de ejecución
- Polimorfismo o paso de mensajes, con enlace dinámico.
- Excepciones

Uniformidad en la OO, único nivel de abstracción

La integración del concepto de proceso en la abstracción de proceso, promueve la definición de una entidad activa, el objeto, y elimina la necesidad de varios niveles de abstracción(objetos, procesos), facilitando la comprensión y utilización del modelo.

Interfaz de alto nivel, independiente de estructuras de bajo nivel

El lenguaje máquina de la arquitectura se basa en la utilización exclusiva de objetos. Es un lenguaje orientado a objetos puro de bajo nivel que será el lenguaje intermedio de la máquina abstracta. Permite declarar clases, definir métodos y manejar excepciones.

Junto con este conjunto de instrucciones de alto nivel, la máquina abstracta propuesta presenta una serie de clases primitivas definidas de forma permanente en el área de clases. Estas clases primitivas forman también parte de la interfaz de alto nivel que la máquina ofrece a las capas superiores ya que algunas operaciones básicas que en las máquinas tradicionales se ofrecen como instrucciones del lenguaje ensamblador, se ofrecen en la máquina propuesta como métodos de clases primitivas, lo que permite ofrecer un conjunto de instrucciones muy reducido.

Juego de instrucciones reducido

Según esto, el conjunto de instrucciones debe ser el mínimo posible, intentando en todo caso que, la introducción de nuevas características no provoque la introducción de nuevas instrucciones.

Se verá más adelante en este mismo capítulo que existe un conjunto de instrucciones mínimo basado fundamentalmente en las instrucciones `new` y `call` que permite añadir a la máquina nuevas e importantes características como la reflectividad sin ampliar el juego de instrucciones, ampliando, eso sí, el conjunto de clases básicas y modificando, en algunos casos, las ya existentes.

Flexibilidad (extensión)

La implantación de la máquina debe hacerse de tal forma que permita actuar sobre sí misma, lo que confiere flexibilidad al entorno integral. Esto se logra, fundamentalmente, haciendo que parte de la funcionalidad de la máquina se ofrezca como objetos normales con los que trabajar mediante referencias y gracias a la definición de un mecanismo de paso de mensajes que permita el paso de control a objetos que implementan parte de la funcionalidad de la máquina.

1.2 Estructura de referencia

A continuación se describen brevemente los elementos básicos en que se puede dividir conceptualmente la arquitectura. El objetivo de esta estructura es facilitar la comprensión del funcionamiento de la máquina y no indica necesariamente que estos elementos existan como tales entidades internamente en una implantación de la máquina.

Los elementos básicos que debe gestionar la arquitectura son:

- **Clases**, que se pueden ver como si estuviesen agrupadas en un **área de clases**. Las clases contienen toda la información descriptiva acerca de las mismas.
- **Instancias**, agrupadas en un **área de instancias**, donde se encuentran las instancias (objetos) de las clases definidas en el sistema. Cada objeto será instancia de una clase determinada.
- **Referencias**, almacenadas en un **área de referencias**. Se utilizan para realizar la invocación de métodos sobre los objetos y acceder a los objetos. Son la única manera de acceder a un objeto (no se utilizan direcciones físicas). Las referencias contienen el **identificador único y global del objeto** al que apuntan (al que hacen referencia). Para la comprobación de tipos, cada referencia será de un tipo determinado (de una clase).
- **Referencias del sistema**. Un conjunto de referencias especiales que pueden ser usadas por la máquina. Se trata de referencias a objetos que realizan alguna tarea especial en el sistema. Entre ellas se incluyen la referencia **this**, que apunta al objeto actual, **sched**, que señala el objeto planificador actual, **root_sched**, raíz de la jerarquía de planificadores en tiempo de ejecución y **ct**, *current thread*, que referencia al entorno de ejecución actual.
- **Jerarquía de clases básicas**. Existirán una serie de clases básicas en el sistema que siempre estarán disponibles. Serán las clases básicas del modelo único de objetos del sistema. En el anexo B se ofrece una descripción detallada de esta jerarquía de clases primitivas.
- **Entorno de Ejecución** Aunque no se trate de un área como tal, ya que se verá que la ejecución se lleva a cabo de forma implícita en la máquina y, por lo tanto, no aparece en la figura de forma explícita, es obligatorio hacer notar la necesaria gestión de la ejecución de los métodos, por medio de objetos internos a la máquina.

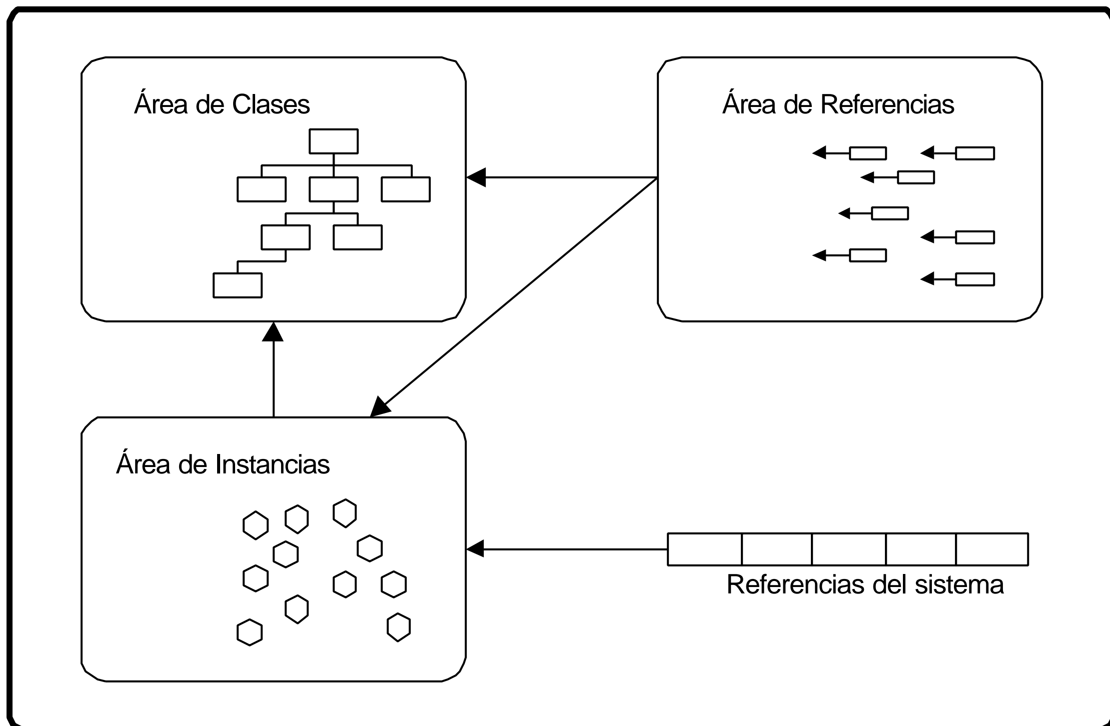


Figura 15.1 Estructura de referencia de una máquina abstracta OO.

1.3 Juego de instrucciones

El juego de instrucciones OO de alto nivel deberá ser independiente de estructuras internas de implantación. Para ello se describirá en términos de un lenguaje ensamblador.

1.3.1 Instrucciones declarativas

La unidad de descripción en este Sistema Integral Orientado a Objeto son las **clases** y por ello, la arquitectura de la máquina abstracta almacena la descripción de las mismas en el **área de clases**.

Para la definición de clases, el lenguaje de la máquina abstracta ofrece un conjunto de instrucciones declarativas, cuyo resultado es la descripción de la información de una clase.

Estas instrucciones sirven para definir completamente y sin ambigüedad una clase, lo que supone identificar la clase de forma unívoca mediante un **nombre** que no esté asociado ya a alguna clase, especificar las **relaciones** entre distintas clases, tanto de **herencia**, como de **agregación** o de **asociación**, y, por último, definir la **interfaz** de la clase, o conjunto de métodos que la clase ofrece al exterior.

1.3.2 Instrucciones de comportamiento

Permiten definir el comportamiento de la clase, es decir, el comportamiento de sus métodos. La descripción del código que compone el cuerpo de un método supone la enumeración ordenada de un conjunto de instrucciones de entre las que se indican a continuación.

- **Instrucciones para manipulación de objetos** a través de las referencias
- **Invocación de método** a través de una referencia. Son, en esencia, la única operación que se puede realizar en un objeto y suponen la ejecución del método invocado. La ejecución de los métodos se representa mediante un objeto interno [1] de la máquina denominado **thread** que, virtualmente, se encuentra almacenado en el **área de threads**.
- **Control de flujo**. Instrucciones para controlar el flujo de ejecución de un método entre las que se encuentran la **finalización de un método**, los **saltos** y las **instrucciones de control de flujo para gestionar las excepciones**.

En el Anexo A se muestran las distintas instrucciones del ensamblador soportadas por la implantación actual de la máquina.

En lo que sigue, nos interesarán únicamente aquellas que se refieren a la invocación de métodos, la finalización de los mismos y las excepciones.

2 Soporte Básico al Modelo de Objetos

La máquina abstracta proporciona el soporte básico a la existencia, la identificación y el comportamiento de los objetos.

2.1 Modelo básico de Objetos

Dentro de la máquina abstracta propuesta, el único elemento existente en tiempo de ejecución es el **objeto** y la máquina define un objeto como una **instancia de una clase en tiempo de ejecución**. Por tanto, la consecución de características como la herencia múltiple o la agregación, dependerá directamente de la forma en que la máquina abstracta represente y gestione las clases.

Además, todo trabajo con objetos se hace siempre a través de una **referencia** al mismo, nunca sobre la instancia propiamente dicha.

2.1.1 Heredero de las Metodologías OO

Si se tiene en cuenta que la principal labor de una arquitectura orientada a objetos es la de dar soporte a los lenguajes orientados a objetos, pero que a su vez la labor de éstos es la de dar soporte a las metodologías de análisis y diseño orientado a objetos, la forma de declarar las clases en la máquina abstracta, podría tomarse de dichas metodologías (OOA, OMT[Rum96], Booch[Boo94], etc…).

[1] Aunque aquí se afirme que los hilos son objetos internos de la máquina, veremos posteriormente en este mismo capítulo que la exposición de los mismos al exterior se hace necesaria para poder dotar a la máquina de una verdadera arquitectura reflectiva.

Aunque con diferente notación, todas las metodologías anteriores coinciden en la representación de los objetos (definiendo en ellos sus variables miembro y sus métodos) y las relaciones que hay entre ellos.

Estas relaciones básicamente se pueden agrupar en tres tipos, y, siguiendo la terminología de OAA, son:

- **General/Específico** o **Herencia** (o *is a*).
- **Todo/Parte** (o *aggregation*). Se refiere al hecho de que, en la composición de una clase, intervengan otras cualesquiera. En el caso de la arquitectura propuesta, en la que no existen tipos básicos, ésta relación incluye también a los atributos de la clase.
- **Asociación** (o *association*). Brevemente, se puede describir como aquella relación entre dos clases que no se puede ubicar en ninguna de las dos anteriores.

2.1.2 Declaración de clases en la arquitectura propuesta

Para cada clase que se quiera definir, será necesario enumerar ciertas características de las clases que la definan completamente y sin ambigüedad.

- **Nombre de la Clase**. En primer lugar, es necesario dar un nombre a la clase. Este nombre deberá ser único y se almacenará, junto con el resto de la información que le siga, en el área de clases.
- **Relaciones con otras clases**. En segundo lugar, se especifican aquellas clases con las que la clase en cuestión guarda alguna relación.
  - **Herencia Múltiple.** Se enumeran todas las clases de las que hereda la clase que se está definiendo, encargándose la máquina abstracta de solucionar los conflictos de variables miembro y métodos con el mismo nombre.

    Como regla general, se accede siempre a la variable o método de la clase que primero aparezca nombrada. En caso de que se desee acceder al otro método, se debe especificar, explícitamente, la clase a la que pertenece antes del nombre del método.

    Toda **derivación** es **virtual**, lo que supone que, si aparece la típica estructura de herencias como la de la Figura 15.2, la clase D solo tiene una instancia de la clase A, de manera que se mantiene sincronizada respecto a los cambios que se puedan hacer desde B y desde C.

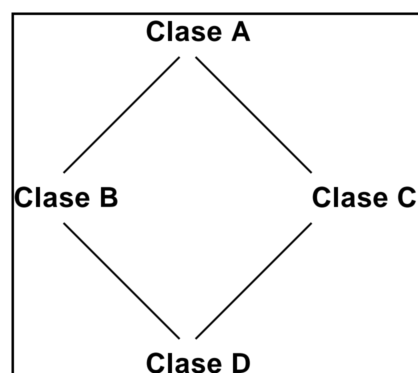


Figura 15.2 Herencia Virtual.

- **Agregación**. Se enumeran los objetos que pertenecen a la clase junto con el nombre que se les da a cada uno. Los objetos agregados se crean cuando se crea el objeto que los contiene y se destruyen cuando éste se destruye. Son equivalentes a declarar en C++ un objeto como variable miembro de otro.
- **Asociación.** Se declaran también los objetos que participan de una relación de asociación con la clase que se está definiendo. A diferencia de los anteriores, estos objetos no se crean automáticamente al crear el objeto ni tampoco se destruyen automáticamente al borrarlo, sino que todo ello es responsabilidad del programador.

  El equivalente en C++ sería declarar un puntero al tipo deseado, que es responsabilidad del programador gestionar (asignarle un objeto, apuntar a otro objeto distinto si la relación se traspasa a otro individuo, destruirlo si es que es nuestra labor hacerlo, etc…)[2].

- **Conjunto de métodos**. Se definen a continuación los métodos que componen la clase. La declaración de estos especificará su nombre, parámetros identificados mediante la clase a la que pertenecen y valor de retorno si es que existe.

2.2 Clase Básica Objeto

Se define clase básica como aquella cuya implantación está codificada internamente en la propia máquina abstracta, utilizando para ello el lenguaje de desarrollo empleado para codificar la máquina.

Dentro de la jerarquía de clases básicas ofertadas por la máquina abstracta, se encuentra la clase **Object**, de la que derivan todos los objetos existentes en el sistema, bien sean de usuario o de otras capas superiores como el sistema operativo, bases de datos, etc.

2.2.1 Nivel de Abstracción Único: Uniformidad en torno al objeto

Existen dos características de la máquina abstracta que garantizan la uniformidad en torno a la abstracción de objeto, o lo que es lo mismo, impiden que la máquina proporcione ninguna otra abstracción en tiempo de ejecución, excepto el objeto.

Juego de Instrucciones

Por un lado, el juego de instrucciones reducido que la máquina abstracta ofrece pocas posibilidades al usuario para añadir elementos nuevos en tiempo de ejecución.

Concretamente, las únicas instrucciones que la máquina ofrece para añadir elementos son la **instrucción declarativa Class** para añadir una clase o bien, la **instrucción new** para crear una instancia de una clase, o sea, un objeto.

Por tanto, las operaciones que la interfaz de la máquina abstracta permite realizar en tiempo de ejecución, se limitan a aquellas referidas a la definición, creación y manipulación de objetos.

---

[2] Lo anterior no quiere decir que para los miembros aggregate se guarde una instancia del objeto y que los para association se guarde un puntero. En Carbayonia **todo se hace a través de referencias** por lo que en los dos casos en realidad se estarán reservando espacio para punteros. Por tanto, lo que Carbayonia expresa mediante estas dos categorías es el *papel* o responsabilidades de la clase respecto a sus componentes, y no como se almacena el mismo dentro de la clase.

Es decir, en tiempo de ejecución, el único elemento que existe es el objeto y, para resolver un problema del mundo real, la máquina abstracta permite crear objetos e invocar métodos en los mismos. No existe ninguna otra abstracción en tiempo de ejecución.

Clase Básica

Por otro lado, la máquina abstracta define un objeto como una instancia en tiempo de ejecución de una clase, creada gracias a la instrucción `new`[3].

Toda clase pertenece a la jerarquía de clases de la máquina, que tiene como raíz la clase Object, por tanto, todo es un Objeto. Esta clase define el comportamiento básico de cualquier nuevo elemento que se cree en el sistema.

La definición de una clase básica de objetos de la que derivan todos los objetos existentes en el sistema, promueve que todos los objetos estén dotados de una serie de características básicas que, ineludiblemente, heredarán.

2.2.2 Identificador único de objetos

Todo objeto que existe en el sistema en tiempo de ejecución es una instancia de clase básica `Object` y se comportará según el comportamiento definido para él por la máquina abstracta, al tratarse esta de una clase básica.

La máquina abstracta, cuando se invoca la instrucción new sobre cualquier clase de la jerarquía de clases de la máquina (sean estas clases básicas o de usuario), crea una nueva instancia de un objeto al que proporciona un identificador único y global en el sistema [ATD+98a, ATD+98b].

2.2.3 Reflectividad: Aumento del Soporte al Objeto por parte de la máquina abstracta

El aumento en la semántica del objeto supone un soporte más sofisticado al objeto. Dicho de otra forma, la representación del objeto que la máquina soporta se ve modificada por la necesidad de mantener la relación entre los objetos y el conjunto de meta-objetos que completan su entorno de ejecución.

Soporte en el entorno de ejecución base al enlace base-meta

Se añade a la clase básica Object, el campo **META**. De esta forma, las instancias en tiempo de ejecución podrían acceder a su meta-espacio simplemente consultando el valor del campo META.

Así, pueden determinar a qué objetos pasarle el control cuando la máquina tenga lagunas en el proceso de ejecución. También se hace posible cambiar el meta-objeto simplemente cambiando el valor asociado a esa posición del objeto.

En esta propuesta concreta, este enlace **META** no será una única entrada sino un conjunto de entradas a los distintos meta-objetos que definen el entorno en tiempo de ejecución del objeto base, a saber, los meta-objetos encargados del paso de mensajes, los meta-objetos encargados del control de la concurrencia y aquel o aquellos encargados de la planificación.

[3] Se puede considerar que la instrucción new es equivalente al envío del mensaje new al objeto clase concreto del que se pretende crear una instancia.

Este punto de entrada al meta-espacio se puede representar a su vez como un objeto, al estilo del reflector de Apertos[Yok92] o Merlin[AK95].

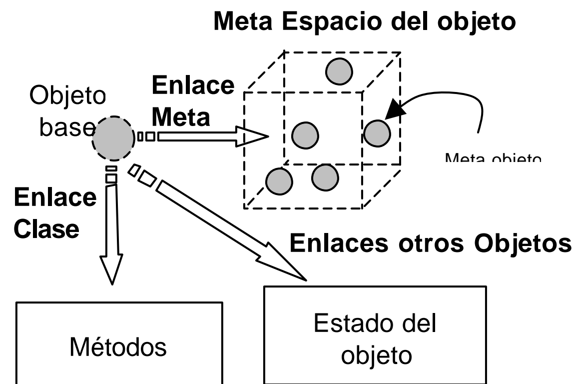


Figura 15.3 Enlace Meta: Asociación del objeto y su meta espacio

2.3 Polimorfismo

El polimorfismo o paso de mensajes, se logra gracias a que, la única posibilidad que ofrece la máquina para ejecutar métodos, es la invocación de los mismos.

Si a ello se suma que es en tiempo de ejecución cuando se resuelve el objeto, método y parámetros de la invocación, da como resultado que el resultado de una línea de código sólo puede ser conocido en tiempo de ejecución.

2.4 Excepciones

Actualmente, las excepciones se consideran ya un elemento más de cualquier sistema de computación por lo que se ha optado por incluir en la propia máquina el tratamiento de excepciones.

3 Soporte Básico a la Concurrencia

En el modelo de objetos propuesto en esta tesis, los objetos se convierten en entidades activas, servidores multhilo que aceptan y procesan los mensajes en un orden decidido por ellos mismos, similares a los actores de [Agh86].

La introducción y gestión de la concurrencia tiene un gran aliado en el modelo de objetos activo que se propone implantar en la máquina abstracta ya que simplifica áreas como la comunicación entre las entidades, la representación de la computación o el control de la concurrencia.

3.1 Objetos Activos como Modelo de Unificación

La máquina abstracta define pues, **objetos activos**. Esto significa que los objetos encapsulan datos y métodos, como los objetos tradicionales. Pero también significa que

los objetos son **entidades autónomas** que encapsulan hilos de control capaces de ejecutar los métodos que componen la interfaz del objeto a petición de otros objetos que lo solicitan.

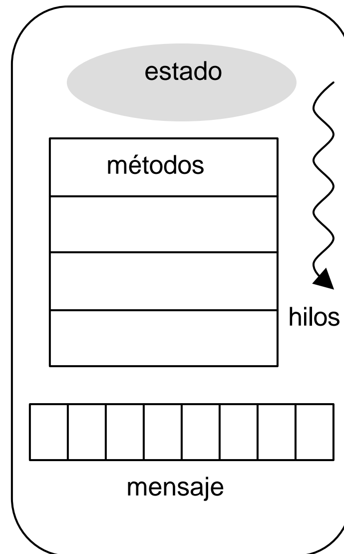


Figura 15.4 Anatomía interna de un Objeto.

Para la implantación de un modelo de objetos activo, la máquina abstracta sigue, fundamentalmente, dos ideas en su diseño: la ausencia de una entidad paralela al objeto y la estrecha relación que la máquina abstracta establece entre un objeto y su implantación en el soporte a la abstracción de objeto.

3.1.1 Ausencia de otras Abstracciones que no sean el objeto

Un modelo de objetos pasivo, supone la existencia de dos abstracciones con semántica y comportamiento diferente.

Por un lado el objeto, que encapsula una entidad del dominio del problema ofreciendo una interfaz bien definida al exterior. Por otro lado, una supra-entidad, el proceso, diferente e independiente del objeto que, aunque se represente mediante un objeto[RJC88], tiene un comportamiento diferente. No encapsula ningún elemento del problema, sino que sirve para representar el flujo de ejecución.

Las diferencias existentes entre ambas entidades son muchas e importantes. Así, por ejemplo, ambas entidades se comunican de forma diferente. Los objetos mediante paso de mensajes, los procesos mediante objetos compartidos. Además, la presencia de entidades con comportamiento diferente provoca la aparición de acciones en el sistema totalmente ajenas al modelo de objetos, como pueden ser el establecimiento de una función de correspondencia entre los objetos y los espacios de direcciones de los procesos.

La máquina abstracta propuesta defiende la ausencia total de cualquier abstracción que no sea el objeto, lo que se asegura ofreciendo una **jerarquía de clases primitivas**, en la que no está presente la clase proceso ni similar, de la que deriva cualquier otra.

LA raíz de esta jerarquía es la clase object y, la máquina abstracta implementa una única instrucción para crear elementos, **new**, que crea instancias de una clase. No existe

por tanto ninguna otra instrucción – fork, NewProcess u otras – que permitan crear otro tipo de abstracciones.

3.1.2 Estrecha relación, implementada a bajo nivel en la máquina abstracta, entre un objeto y su computación

La máquina abstracta da soporte a la abstracción de objeto activo, haciendo innecesaria una abstracción nueva, tipo proceso.

Para ello, la máquina **asocia la computación a los objetos** estableciendo una estrecha relación entre un objeto y su computación, mediante el soporte de una abstracción de objeto más compleja que en el caso de un modelo de objetos pasivo, donde el objeto comprende datos y métodos y existe una abstracción tipo proceso, nueva y totalmente independiente del objeto.

Esta estrecha relación entre un objeto y la ejecución de sus métodos se basa en las siguientes premisas.

Objetos internos

La ejecución de un método en un objeto se representa mediante objetos internos de la máquina abstracta.

Relación objeto – ejecución

Estos objetos internos forman parte de la representación en tiempo de ejecución que la máquina ofrece de la abstracción objeto activo. Por tanto, cada objeto conoce, o es capaz de conocer, qué métodos se están ejecutando y cómo, y establecer políticas de sincronización en la ejecución de los mismos.

Paso de mensajes

Por último, los objetos activos interaccionan con su entorno, representado por otros objetos activos, mediante la creación de nuevos objetos y el envío y recepción de mensajes. Para ello, la máquina abstracta proporciona, en su conjunto de instrucciones las instrucciones **new** y **call** y **send**, respectivamente.

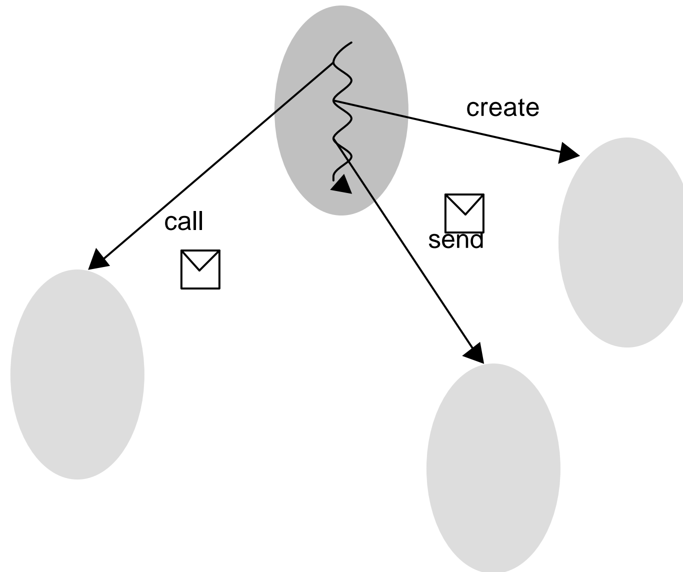


Figura 15.5 Operaciones Primitivas en el modelo: Crear, Enviar Mensajes

La recepción de un mensaje inicia la ejecución del método especificado como argumento de la instrucción **call** o **send**. En respuesta a un mensaje, un objeto puede enviar más mensajes, crear otros objetos y cambiar su estado interno.

Una vez que un método comienza a ejecutarse, continúa hasta que termina o suspende su ejecución por alguna razón.

Estas tres premisas, garantizan que, cara al exterior, los objetos funcionen como una abstracción uniforme.

3.2 Contexto de Ejecución de los Objetos

A continuación se ofrece una descripción más detallada del soporte que la máquina abstracta da a la abstracción de objeto activo, fundamentalmente, en lo que se refiere a cómo se representa su computación y cómo se establece la relación entre el objeto y la ejecución de sus métodos.

3.2.1 Representación en Tiempo de Ejecución de los Objetos por parte de la Máquina Abstracta

Esta representación incluye el **estado interno** del objeto, los **métodos** que manipulan esos datos – representados por la clase a la que pertenece el método – y que los objetos ofrecen como interfaz al exterior, y el **área de ejecución de los objetos**, representación dinámica del **contexto de ejecución** de los objetos, es decir, la representación en tiempo de ejecución de la parte de computación de los objetos.

Contexto de Ejecución

El contexto de ejecución de un objeto comprende toda aquella información que determina en qué estado está la ejecución de los métodos invocados en ese método.

Supone almacenar información acerca de los métodos en ejecución, describiendo completamente y sin ambigüedad el estado actual de la ejecución de tales métodos, así como los métodos cuya ejecución, por alguna razón, ha sido suspendida.

Además de todos aquellas métodos que, aún habiendo sido invocados, su ejecución ha sido retrasada por alguna razón y se encuentran pendientes de servicio.

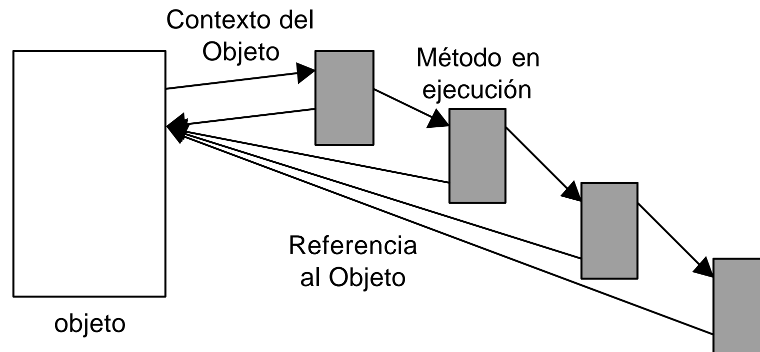


Figura 15.6 Estructura del Entorno de Ejecución del Objeto

Exposición del Engine de Computación

Más adelante en este mismo capítulo se verá que, para que el sistema operativo pueda cooperar con la máquina para construir un entorno de computación adecuada, esta debe exponer al exterior el contexto de ejecución de los objetos, característica interna que será visible al sistema operativo.

Por tanto, la máquina abstracta no sólo da soporte a la ejecución de métodos, sino que expone un modelo de su maquinaria interna de ejecución, del término inglés *engine*, accesible y manipulable para los objetos del sistema operativo.

3.2.2 Descripción de la ejecución de un método

Para dar soporte a la ejecución de métodos, la máquina abstracta define **objetos internos**, es decir, objetos de la propia implantación de la máquina abstracta que describen, en tiempo de ejecución, la ejecución de un método.

Estos objetos describen el contexto de ejecución de un método o lo que es lo mismo, describen el estado actual del servicio de la ejecución de un método en un objeto y, en lo que sigue, los denominaremos **hilos**[4].

Por tanto, el contexto de ejecución de un objeto estará compuesto por un conjunto de objetos internos y la descripción en tiempo de ejecución del objeto incluirá un **área de ejecución del objeto**, conjunto de hilos que representan los métodos en ejecución del objeto.

[4] Aunque se haya elegido el nombre de hilos por analogía con otros sistemas, no debe confundirse con el concepto tradicional de hilo. En este caso, son objetos internos de la máquina que forman parte de los muchos objetos internos que la máquina abstracta utiliza para representar un objeto en tiempo de ejecución.

Estructura del hilo

Cada hilo mantiene información como la siguiente acerca de la ejecución de un método o conjunto de métodos en un objeto.

- Referencia al objeto **instancia** al que está asociado mediante la ejecución de alguno de sus métodos.
- **Referencia al origen de la llamada**, es decir, al hilo en cuyo contexto se invocó al método en ejecución.
- **PC virtual**, que indica la siguiente instrucción del método a ejecutar.
- **Estado**. Define el estado de la ejecución del método. Un hilo puede estar, en cada instante, en uno de los siguientes estados: **Creado**, supone la invocación de un método que todavía no se tiene en cuenta para ejecución. Esto puede suceder por razones muy diversas como una sobrecarga; **Listo**, el método invocado será comenzado a ejecutar tan pronto como se le asigne la máquina; **En Ejecución**; **Suspendido**, con el significado de que, por alguna razón, el hilo no es ejecutable en este momento.
- **Pila**, formada por un conjunto de capas, que se apilan ante llamadas a métodos del mismo objeto o ante la instalación de manejadores de excepciones y se desapilan al retornar de cualquiera de ellos. Cada capa contiene referencias a objetos locales a la llamada y una serie de referencias especiales como **referencia de retorno** o **rr** que referencia al objeto retorno de la capa superior llamada desde la actual y **exc**, **referencia de excepción**, que referencia a un objeto excepción.
- Otra información de estado temporal.

3.2.3 Creación y Destrucción de Hilos

Los hilos abstraen el poder de ejecución de la máquina abstracta, puesto al servicio de los objetos que necesitan ejecutar un método, además de describir todo el proceso de ejecución de un método.

Por tanto, se crearán cuando se invoque un método y se destruirán cuando la ejecución del mismo termine, bien sea con éxito o con algún tipo de error, excepción hecha de la invocación de métodos en objetos primitivos o instancias de clases primitivas, que no necesitan de hilos para la ejecución de sus métodos.

En cualquier caso, es la propia máquina abstracta la que crea y destruye los hilos de un objeto, sin intervención directa por parte del mismo, lo que confiere eficiencia al sistema.

La creación de un hilo tiene lugar cuando un objeto recibe una petición de servicio de un método o lo que es lo mismo, cuando un objeto invoca un método de otro, la máquina abstracta crea un nuevo hilo en el segundo para ejecutar el método solicitado. Por tanto, aunque no haya una intervención directa por parte de los objetos, sí es necesaria su actuación para la creación de hilos ya que esta se produce bajo demanda, al ejecutar cualquiera de las instrucciones de invocación de métodos.

Cuando la ejecución de un método termina, la máquina abstracta retorna el resultado, si existe, al hilo desde el que se invocó al método, instanciando su referencia rr o exc, al valor adecuado. En el último momento, destruye el hilo.

Decisión de Implantación

El hecho de que se defina un **área de ejecución del objeto** en el que existen objetos denominados hilos que se crean y destruyen para cada invocación, no debe interpretarse como una indicación de la implantación que se va a hacer.

Es posible implantar un objeto dotándolo de un pool de hilos que se mapean con estos hilos internos que se definen o utilizar un único hilo, distinguiendo en cada instante qué parte del contexto de ejecución de dicho hilo corresponde a la ejecución en cada objeto de forma que este y sólo este, tenga derecho a examinarla, o incluso a modificarla. Pero esto es una decisión de implantación.

3.2.4 Relación objeto-hilos

Lo realmente importante es que cada objeto sea capaz de conocer, identificar y manipular sus hilos, entidades abstracta que encapsulan la ejecución de un método dentro de un objeto.

La trascendencia de este hecho, identificada ya en otros trabajos como [HK93] radica en las siguientes razones.

- **Distribución**. La definición del Sistema Operativo como un sistema distribuido, implica la, más que probable, invocación de métodos en máquinas remotas. Por tanto, a nivel de aplicación, en una invocación a un método se van a ver involucradas distintas máquinas.

  Por ello, si se desea representar la cadena de invocación como un único hilo, será necesario añadir algún código adicional para mantener la información de las diferentes máquinas involucradas.

- **Seguridad e Integridad**. Es necesario limitar el modo en que un objeto puede manipular a otro. Supóngase que, desde un objeto O, se invoca un método en un objeto O' (digamos, un servidor de ficheros). En modo de depuración, O puede intentar parar todas sus ejecuciones y analizarlas. Sin embargo, no debería parar los hilos en el objeto O'. Es decir, las acciones sobre un hilo deberían afectar únicamente al comportamiento del hilo asociado al objeto que realiza la acción.

Por tanto, se puede crear un objeto interno superior que englobe a distintos hilos que ejecuten una cadena de llamadas en distintos métodos, se puede definir un objeto interno menor que encapsule la ejecución de un método, pasando el hilo a ser la entidad que representa la cadena de llamadas en distintos objetos. Pero todo ello es una cuestión de implantación.

3.3 Concurrencia en el Modelo de Objetos soportado por la Máquina Abstracta

La concurrencia se define en el modelo de objetos de la máquina como **la posibilidad de ejecutar varios métodos simultáneamente**, lo cual será realmente posible en caso de que se disponga de varios procesadores físicos, o se simulará, mediante el reparto de la máquina entre los distintos métodos en ejecución de los distintos objetos.

La máquina abstracta propuesta introduce la concurrencia a dos niveles, requisitos ya descritos en el Capítulo IX.

Por un lado, la **concurrencia entre objetos** gracias a la posibilidad que ofrece la máquina – que no introduce ninguna restricción en este sentido – de que uno o varios, incluso todos, los objetos existentes en el sistema tengan asociados hilos activos simultáneamente.

Por otro lado, la **concurrencia dentro de los objetos**, gracias a la posibilidad que ofrece la máquina – que tampoco introduce ninguna restricción en este sentido – de que cada uno de los objetos existentes en el sistema, tengan asociados varios hilos activos simultáneamente. En este sentido, otros sistemas como Apertos[Yok92], solo permiten un método activo por objeto.

En cualquier caso, la ejecución simultánea de varios métodos necesita de algún mecanismo de control que permita el acceso controlado al estado del objeto, así como coordinar el uso de un objeto por hilos concurrentes[PA97].

La concurrencia en la máquina abstracta, basada en la posibilidad de la ejecución simultánea de varios métodos de un mismo o distintos objetos, depende de dos factores: el **mecanismo para introducir la concurrencia en el modelo** y el **mecanismo de gestión de la concurrencia**.

3.3.1 Introducir la concurrencia en el Sistema

La introducción de la concurrencia se lleva a cabo, principalmente a través de la creación de un nuevo hilo dentro del objeto[5] para procesar el mensaje, aunque un objeto también puede ejecutar métodos de forma espontánea (sin solicitud por parte de nadie) como puedan ser aquellos que recolectan información acerca del estado del sistema, para tomar decisiones y adaptarse al entorno.

La creación de nuevos hilos por parte de la máquina abstracta, se lleva a cabo asociado siempre con un paso de mensaje o **invocación a método**. Para ello, la máquina abstracta ofrece mecanismos de **invocación de métodos síncrono**, en el que el nivel de concurrencia permanecería inalterado, ya que se suspendería la ejecución del método original, y **wait-by-necessity**, en el que ambos métodos se servirían en paralelo, aumentando así el grado de concurrencia en el sistema.

Es decir, la máquina ofrece mecanismos de paso de mensaje para la **introducción controlada de la concurrencia**. Un mensaje no siempre supone aumentar el nivel de concurrencia, lo que sólo sucedería en el caso de que la invocación siguiese la semántica wait-by-necessity.

Más adelante en este capítulo, se verán los mecanismos de paso de mensaje implantados por la máquina abstracta y la concurrencia que se deriva de ellos.

3.3.2 Mecanismos de Gestión de la Concurrencia

Al proporcionar la máquina abstracta la posibilidad de ejecutar en paralelo múltiples métodos de uno o varios objetos simultáneamente, creando para ello los hilos pertinentes en la ejecución de la instrucción de paso de mensajes, es necesario establecer un mecanismo de control flexible que permita asegurar un orden en la ejecución de los métodos.

---

[5] Nuevamente, no confundir el objeto interno hilo definido en el diseño de la máquina con otros posibles hilos de ejecución utilizados en la implantación de la misma.

Control Entre Objetos

El mecanismo más usual de control entre objetos es el paso de mensajes. La necesaria sincronización impuesta por cualquiera de ellos, facilita la implantación de políticas más complejas de gestión de la concurrencia. Esto se tratará en este mismo capítulo, en el apartado 4.

Control dentro de los objetos

Dentro del objeto, el control de la concurrencia tomará dos vertientes distintas:

- Mecanismos de Grano Fino con objetos tipo cerradura.
- Mecanismos de Grano Grueso mediante la colaboración con el Sistema Operativo.

Esto se tratará en este mismo capítulo, en el apartado 6.

4 Comunicación entre objetos

Dado que el sistema integral Orientado a Objetos formado por el sistema operativo + la máquina abstracta garantiza la uniformidad conceptual en torno a la entidad de objeto, la invocación de métodos es la única forma de comunicación permitida. En esto coincide con muchos otros sistemas como Merlin[AK95] o Apertos[Yok92].

La invocación a métodos, o paso de mensajes entre objetos, no sólo es el mecanismo básico de comunicación entre los objetos. El modelo de paso de mensaje utilizado determinará también, en gran medida, el nivel de concurrencia obtenido ya que la invocación de un método en un objeto supone la creación de un hilo de actividad virtual independiente, ligado a ese objeto, para ejecutar el código del método, al igual que se hace, por ejemplo, en CORBA, en cuyo caso se denomina **method activation**[OMG97].

Este hilo se destruirá después de haber llevado a cabo toda la computación indicada en el método y de haber puesto el resultado a disposición del cliente. Los hilos que virtualmente se crean para servir peticiones en los objetos, se consideran también parte del estado del objeto[Hau93].

Sin embargo, la creación de nuevos hilos virtuales en los objetos no siempre supone un aumento en el grado de concurrencia porque, dependiendo del modelo de paso de mensajes utilizado, puede suponer o no la activación de un contexto más simultáneamente. Lo cual además, posibilita la sincronización entre el objeto que realiza y el que recibe la llamada.

Sin embargo, y paralelamente al mayor grado de concurrencia que introducen y por ende al aumento del rendimiento, cada uno de los modelos de comunicación presenta una complejidad inherente.

4.1 Instrucción de la máquina

La comunicación entre objetos se lleva a cabo a nivel de la máquina abstracta mediante la adición de las instrucciones **call** y **send** al juego de instrucciones.

Estas instrucciones desencadenan todas las acciones necesarias para informar al objeto destino de la invocación de un método por parte de otro objeto. Así mismo, también realiza las acciones pertinentes para que el objeto origen se comporte acorde a

la semántica definida por el modelo, bien sea esta continuar la ejecución normal o suspenderla.

Sin embargo, la ejecución efectiva del método no tiene por qué ser inmediata. Queda en manos del Sistema Operativo decidir cuál es el momento más adecuado para comenzar la ejecución del método.

Se ve así que, mientras el Sistema Operativo planifica, la máquina abstracta se limita a ofrecer los mecanismos de bajo nivel.

Enlace Estático y Dinámico

La implantación de las instrucciones de paso de mensajes permite pasar los argumentos como referencias a los valores efectivos que tal argumento debe tomar o como referencias a objetos, desconocidos en tiempo de compilación, que serán resueltas en tiempo de ejecución.

Así, es posible conocer el nombre del método a invocar, los argumentos e incluso el objeto en tiempo de compilación o no.

- **Método**. Si el nombre no es conocido en tiempo de compilación, el argumento NombredelMétodo de la instrucción call/send, se sustituye por un objeto string que tomará valor en tiempo de ejecución.
- **Argumentos**. Dado que los argumentos se pasan como un array, no hay diferencia en la actuación.

Objeto. Si el objeto no es conocido en tiempo de compilación, se puede pasar una referencia a un objeto que, posteriormente, se resolverá al objeto indicado.

4.2 Modelos de Comunicación

En Carbayonia se ha optado por introducir dos modelos de paso de mensaje[Car89, Car93], a saber, el **síncrono** por medio de la instrucción **call** y el **wait-by-necessity**, con la instrucción **send**, de forma similar a otras máquinas abstractas y sistemas. Estos modelos se pueden ver como una invocación de métodos y un paso de mensaje, respectivamente.

Por ejemplo, Apertos[Yok92] implementa las primitivas **Call** que invoca un método en un objeto destino que se activa directamente y **Send**, en cuyo caso, el objeto origen continúa su ejecución.

Así mismo, distintas implantaciones de CORBA ofrecen mecanismos de paso de mensajes o invocación asíncrona, además del tradicional paso de mensajes síncrono. En concreto, casos como Expertsoft[Exp95] o Voyager[Obj99] implementan un mecanismo muy similar al que se propone en la máquina abstracta que se describe aquí.

Sea cual sea el modelo de invocación utilizado, es necesario definir también cuáles serán las acciones a llevar a cabo para retornar los resultados y, en caso necesario, desbloquear las acciones necesarias.

4.2.1 Modelo Síncrono: Call

Aunque la invocación de un método en un objeto destino supone la creación de un hilo en el mismo, no existe un incremento del grado de concurrencia en el sistema, ya que el hilo origen de la llamada se suspende hasta que le retornen el resultado.

La implantación de este modelo puede hacerse de forma explícita, añadiendo a la máquina abstracta las clásicas primitivas de comunicación send y receive. Sin embargo, es posible también hacerlo de forma implícita, de forma que sólo sea necesario añadir una instrucción de envío a la máquina.

La Máquina Abstracta implementa a bajo nivel la instrucción **Call** que desencadena las mismas acciones que un envío de mensaje, activando la ejecución del método solicitado en el objeto llamado y transmitiendo el control de un objeto a otro.

La máquina abstracta crea un nuevo hilo en el objeto destino, que representa la ejecución de un nuevo método. A continuación transmite el control al objeto destino para ejecutar el método solicitado, bloqueando el hilo de ejecución origen de la llamada y cediendo el tiempo de ejecución del objeto origen al objeto destino, en un símil del hand-off[Bla90].

Así, se hace innecesaria la instrucción *receive* en el objeto destino ya que la propia máquina se encarga de activar un nuevo contexto en el objeto destino.

Retorno de Resultados

Es la propia máquina abstracta la que se encarga de devolver los resultados cuando la ejecución termine. Con esto se evita tener que introducir primitivas de comunicación explícitas tipo **wait** y **reply** para el retorno de los resultados, consiguiendo con ello una máquina tipo RISC más sencilla.

La máquina abstracta lleva a cabo el retorno de resultados destruyendo el hilo del objeto destino y desbloqueando el hilo origen de la llamada, que se hallaba esperando la respuesta.

El modelo de devolución de resultados síncrono se basa en que, en cada hilo existe una referencia especial, la **rr (referencia de retorno)** donde la máquina se encarga de guardar el resultado esperado. La máquina abstracta se encarga de poner en **rr** el valor correspondiente, y activar al hilo origen de la llamada, es decir, ponerlo listo para ejecutar.

También es posible retornar, no un objeto resultado, sino una excepción, en cuyo caso se sigue exactamente el mismo proceso, excepto que, en lugar de instanciar la referencia rr, la máquina pone en la referencia **exc** (**referencia excepción**), el objeto excepción correspondiente.

Cuando el hilo reanude su ejecución en el contexto concreto que realizó la llamada, la máquina puede inspeccionar la referencia de retorno identificando un retorno normal o una excepción, según sea el caso.

4.2.2 Modelo Wait-By-Necessity: Send

En este caso, la máquina abstracta crea un nuevo hilo de ejecución en el objeto destino, que se encargará de realizar la ejecución del método solicitado y devolver la respuesta.

Sin embargo, en contra de lo que se hacía en el caso síncrono, la máquina abstracta ni bloquea el hilo origen, ni cede su tiempo de ejecución al nuevo hilo de ejecución del objeto destino. Ambos métodos se ejecutarán de forma concurrente y la sincronización tendrá lugar, si es necesaria, cuando el objeto llamador solicite un servicio del objeto resultante de la ejecución del método llamado.

Objetos Virtuales

Cuando se invoca un método con la instrucción **Send** que debe devolver un objeto con una determinada referencia, la máquina abstracta crea una instancia con esa referencia. Esta instancia será un objeto de la clase **VirtualObj**, de forma que, cuando el método termine y devuelva el objeto resultado, la máquina abstracta sustituirá el objeto virtual por el objeto resultado real.

Al disponer de una referencia al objeto resultante, se pueden realizar operaciones con ella, como invocar un método en el objeto asociado a la referencia. Ahora bien, si se invoca un método en el objeto resultado, bien sea con `Call` o con `Send`, y la máquina abstracta se encuentra con que el objeto es un objeto virtual, es decir, el objeto resultante todavía no ha sido instanciado, bloqueará el hilo de ejecución que realiza la invocación y lo registrará en el objeto virtual.

Al terminar el método que tenía que crear el objeto, antes de eliminar el objeto virtual, despertaremos a todos los métodos que intentaron acceder al objeto que ya estará creado y que deberán volver a realizar la invocación al método.

Objetos Virtuales Transparentes

Los objetos virtuales son totalmente transparentes. Se implementan internamente en la máquina abstracta por lo que, a alto nivel, no existen y es imposible distinguir un objeto normal de uno virtual.

Retorno de Resultados

El retorno de resultados es bastante simple y se produce cuando se instancia el objeto resultado.

La referencia correspondiente a este objeto podría tener ya invocaciones esperando. En este caso, se desbloquean los hilos y el objeto resultado puede comenzar a servir los métodos solicitados.

El desbloqueo de las invocaciones en espera, se lleva a cabo de forma automática. La máquina abstracta puede encargarse de hacerlo, de paso que crea el objeto resultado.

A continuación se ilustra el funcionamiento.

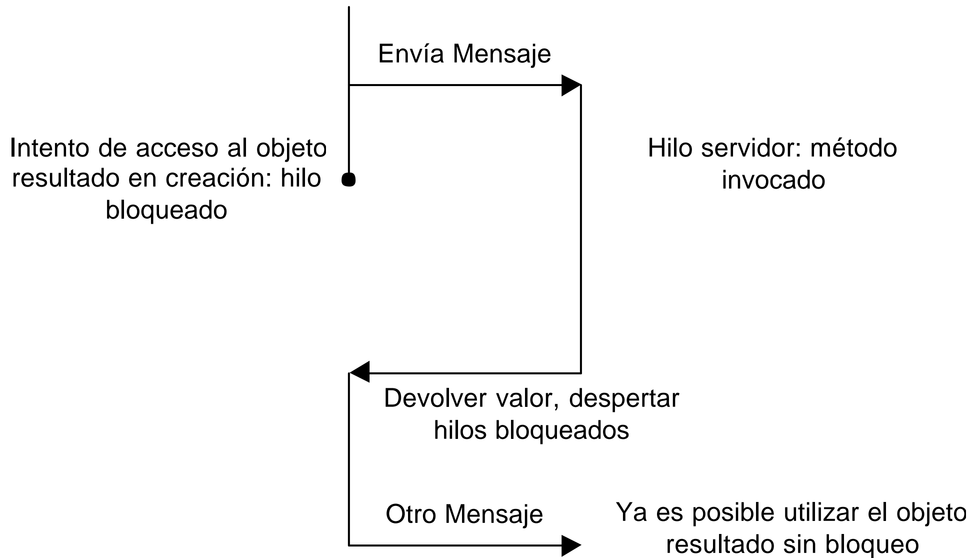


Figura 15.7 Invocación Asíncrona

De esta forma se aprovecha la concurrencia inherente a la instrucción Send liberando al programador e incluso al compilador, de la complejidad añadida que supone el asincronismo.

4.3 Reflectividad del Comportamiento

La reflectividad del comportamiento propuesta en esta tesis, se sustenta en gran medida sobre el mecanismo de paso de mensajes definido por la máquina abstracta.

En primer lugar, dado que los meta-objetos son objetos de primera clase, debe ser factible comunicarse con ellos vía paso de mensajes.

En segundo lugar, el paso de mensajes se definió en el capítulo XIV como uno de los eventos que provoca el paso de control del nivel base al meta-nivel.

Las instrucciones de paso de mensaje, call y send, cumplen pues una doble función en el SIOO. Por un lado, permiten añadir y controlar la concurrencia en el sistema, además de ser el mecanismo de comunicación entre los objetos del sistema. Por otro, trascienden de su carácter de primitiva de comunicación a bajo nivel, para convertirse en el punto de inflexión para conseguir la reflectividad del comportamiento en el sistema.

Invocar un método de un objeto del nivel base por parte de otro objeto del nivel base, ya sea mediante la primitiva call o send, provoca el paso del control al meta nivel del objeto. Son por tanto, *traps* al sistema operativo.

Todo ello hace necesario adaptar el mecanismo de paso de mensajes a la introducción de la reflectividad, modificando para ello las instrucciones de la máquina.

4.3.1 Paso de control al meta-nivel

Cuando un objeto del nivel base invoca un método en otro objeto del nivel base, la máquina abstracta, en lugar de definir por sí misma la semántica del paso de mensajes,

pasa el control al meta-objeto correspondiente (reflectividad implícta), que realizará, como parte del paso de mensaje, las acciones definidas por el usuario.

De esta forma, se permite al usuario personalizar su entorno. El comportamiento específico, depende del meta-objeto en particular (véase capítulo XIV).

Como Call y Send son, a la vez que el mecanismo básico de comunicación, uno de los medios de transferencia de control del nivel base al meta-nivel y viceversa, se puede distinguir en estas instrucciones, un caso base y un caso reflectivo.

4.3.2 Envío de mensajes: Ejecución de la Instrucción Call por parte de un objeto

Caso Reflectivo

El caso reflectivo del envío de mensajes se produce cuando ambos objetos involucrados en la transmisión son objetos del nivel base. Se trata aquí de reflectividad implícita.

En este caso, la instrucción Call funciona como sigue:

1. **Transición del nivel base al meta-nivel**. El control se pasa al meta-objeto Emisor que pertenece al meta espacio del objeto del nivel base correspondiente. Esta transferencia se lleva a cabo mediante la invocación del método **RCallE** en este meta objeto. Un poco más adelante se trata el caso de la ejecución de un método de un meta objeto.
2. Esta transferencia es **síncrona**, por lo que la máquina abstracta bloquea el hilo en el objeto del nivel base que realiza la llamada hasta que la ejecución del método RCallE termine y devuelva el control.
3. Se llevan a cabo las **actividades en el meta-espacio** que se especificarán en el Capítulo siguiente.

   Estas acciones pueden incluir llamadas a otros objetos y, evidentemente, incluyen la transferencia de control al objeto del nivel base destino de la llamada para la ejecución efectiva del método.

De esta forma se mantiene la compatibilidad con la semántica de la instrucción Call. El objeto del nivel base no continúa hasta que la ejecución del método invocado haya terminado.

Caso Base

Este caso se presenta en tres ocasiones.

- **Invocación de un método en un objeto primitivo**. Es la más evidente de todas ellas. En este caso, no se permite ningún tipo de reflejo de la computación, básicamente, por cuestiones de eficiencia.
- **Transferencia de control del nivel base al meta-nivel o Reflectividad Explícita**. El segundo de los casos se presenta cuando se invoca un método de un meta objeto desde un objeto del espacio base.
- **Transferencia de control del meta nivel al nivel base**. El tercer caso se presenta justo cuando la invocación se produce al contrario, cuando se invoca un método de un objeto del nivel base desde un meta objeto.

La máquina abstracta actúa de meta espacio básico, finalizando así la recursión reflectiva.

La gestión del caso base es sencilla y supone la ejecución del método invocado.

1. Si se trata de un método en un objeto primitivo o de un método en un meta objeto, se ejecuta inmediatamente el método en cuestión, utilizando para ello el mismo hilo de control desde el que se realiza la invocación, lo que supone una considerable mejora en el rendimiento. La ejecución de estos métodos se hace contando con el tiempo de la máquina cedido por el objeto que invocó el método.
2. Si se trata de una transferencia de control del meta nivel al nivel base, tampoco se refleja la computación, ya que este es el último paso para la ejecución efectiva del método.

a) Si el objeto destino de la invocación está instanciado, la ejecución del método correspondiente en el objeto base se realiza síncronamente con el meta objeto. Se crea un nuevo hilo de control virtual en el objeto del nivel base y se comienza la ejecución del método. Hasta que este nuevo hilo no termine, no devuelve el control al meta nivel.

   La ejecución de este nuevo hilo de control utiliza el tiempo de procesador cedido por el objeto origen de la invocación. la máquina abstracta bloquea al hilo que realizó la invocación y activa, directamente, sin pasar por el planificador, el hilo nuevo para la ejecución del método, para mayor eficiencia.

b) Si el objeto destino de la invocación no está instanciado, se bloquea el hilo origen de la llamada, insertándolo en la lista de hilos parados en espera de que el objeto virtual se instancie, en cuyo caso, se reanuda su ejecución.

   El bloqueo de un método, en general, la parada por cualquier motivo del método en ejecución, provoca la activación del planificador correspondiente, mediante la invocación de uno de sus métodos, con el fin de activar un nuevo hilo de ejecución.

4.3.3 Envío de mensajes: Ejecución de la Instrucción Send por parte de un objeto

Caso Reflectivo

El caso reflectivo del envío de mensajes se produce también cuando ambos objetos involucrados en la transmisión son objetos del nivel base. Se trata aquí de reflectividad implícita.

El funcionamiento de Send sigue estos pasos básicos:

1. **Transición del nivel base al meta-nivel**. Igual que en el caso de la instrucción Call, se cede el control de la máquina al meta-objeto Emisor. En este caso, la transferencia se lleva a cabo mediante la invocación del método **RSendE** en este meta objeto.
2. Igual que antes, esta transferencia es **síncrona**, por lo que la máquina abstracta bloquea el hilo en el objeto del nivel base que realiza la llamada hasta que la ejecución del método RCallE termine y devuelva el control.

3. Se llevan a cabo las **actividades en el meta-espacio** que se especificarán en el Capítulo siguiente.

   Igual que en el caso de la instrucción Call, estas acciones pueden incluir llamadas a otros objetos aunque en este caso no incluyen la transferencia de control al objeto destino para la ejecución efectiva del método, sino que supondrán la creación de un nuevo hilo de control en el objeto destino, pero no su ejecución inmediata.

De esta forma se mantiene la compatibilidad con la semántica de la invocación wait-by-necessity. El objeto del nivel base continúa una vez que se haya asegurado de que se han llevado a cabo todas las acciones necesarias para la ejecución del método destino.

Caso Base

Igual que en el caso de la instrucción Call, se produce en tres ocasiones. Cuando se invoca un método en un objeto primitivo, cuando se invoca un método de un meta objeto desde un objeto del nivel base o cuando se invoca un método del nivel base desde el meta nivel. También en este caso, la máquina abstracta actúa de meta espacio básico.

La gestión del caso base es sencilla y supone la ejecución del método invocado.

1. Si se trata de un método en un objeto primitivo o de un método en un meta objeto, se procesa igual que si se tratase de la instrucción Call.
2. Si se invoca un método de un objeto del nivel base desde un objeto del meta nivel, se crea un nuevo hilo en el objeto destino para ejecutar el método correspondiente.

   Sin embargo, no se comienza su ejecución inmediatamente, sino que el control vuelve al meta objeto.
3. Invocar al planificador, informándole de la existencia de un nuevo hilo de control.
4. Acciones del meta-nivel posteriores a la ejecución efectiva de la llamada.

Es consistente con la semántica wait-by-necessity, no esperando la ejecución del método invocado para devolver el control.

5 Mecanismo de excepciones

La gestión de excepciones es el mecanismo que proporcionan máquina abstracta y sistema operativo para permitir a las aplicaciones el **tratamiento metódico y ordenado de los errores** y evitar, en lo posible, que estos provoquen la finalización incorrecta de las mismas.

El área de la gestión de excepciones cambia sustancialmente con la introducción de la concurrencia y la posibilidad de la computación distribuida frente a un entorno puramente secuencial y está muy influenciada por el mecanismo de invocación de métodos que se utilice.

Referencias para la gestión de Excepciones

En el contexto de la ejecución de un método se definen las referencias **rr** (*return reference*) y **exc** (*exception*), en las que se retorna la referencia de retorno, si existe, o una excepción, si existe.

La referencia `exc` se puede utilizar para pasar un objeto que represente la causa de la excepción.

5.1 Las instrucciones

Se definen dos instrucciones para el control de las excepciones: Handler y Throw.

```
Handler Etiqueta
...
Throw
```

La misión de la instrucción Handler (manejador) es indicar la dirección donde debe continuar el flujo de ejecución, en caso de que ocurra una excepción (la dirección del manejador). La dirección será aquella que corresponda a la posición de la etiqueta en el código fuente.

Throw (lanzar) lanza una excepción. La ejecución de esta instrucción provoca, en primer lugar que se cree una instancia de un objeto excepción y se asocie con la referencia **exc** (*exception*) de la capa de contexto actual. Como en el caso del valor de retorno de un método, se podría utilizar una manera alternativa que consista en que el objeto causa de la excepción sea un parámetro del Throw.

En segundo lugar, la ejecución del programa continuará en la dirección del último manejador ejecutado. En caso de que no haya ninguno la ejecución se dará por finalizada. Los distintos manejadores se van apilando de manera que tienen prioridad los últimos que se hayan ejecutado.

El lanzamiento de una excepción puede ser por un posible error en tiempo de ejecución (salirse del rango de un array) o bien porque el usuario la haya lanzado con la instrucción Throw.

5.2 Instalación de un Manejador para la excepción

La **gestión de una excepción** consta de dos partes: **instalación de un manejador** y **ejecución del mismo**.

Instalar un gestor para la excepción

Un gestor no es más que un contexto en un hilo de ejecución, con la información necesaria para poder gestionar la excepción.

La instalación de gestores para las excepciones que puedan producirse, es precisamente la función de la instrucción de la máquina `Handler <etiqueta>`.

La ejecución de dicha instrucción hace que la máquina abstracta almacene en la pila del hilo en ejecución un contexto de excepción, que indica la dirección de un gestor adecuado a la excepción que se produjo, es decir, la dirección donde se debe continuar el flujo de ejecución en caso de que ocurra una excepción.

La dirección será aquella que corresponda a la posición de la etiqueta en el código fuente. Y así sucesivamente con todos los `handler` que aparezcan.

Los manejadores se extraen cuando se produce una excepción y también se descartan cuando se sale del método en el que fueron declarados sin que se produjese una excepción.

5.3 Ejecución del Manejador para gestionar la excepción producida

Cuando se produce una excepción, es necesario ceder el control a un manejador que la gestione.

Un objeto puede instalar o no manejadores para distintas excepciones en sus métodos. Estos gestores se van apilando en la pila de contextos del objeto, de manera que tienen prioridad los últimos que se hayan ejecutado, instalándose más cerca de la cima de la pila.

Ante la presencia de una excepción, un objeto puede resolverla él mismo o propagarla hacia atrás en la cadena de llamada/respuesta. El flujo de control a partir de este momento dependerá de si el método en ejecución tiene o no instalado un gestor para la excepción correspondiente.

En cualquier caso, es necesario encontrar un gestor para la misma, aunque para ello sea necesario retroceder en la cadena de llamadas a métodos.

Si no se encuentra ningún gestor para manejar la excepción, en toda la cadena de llamadas, la ejecución se dará por finalizada.

5.3.1 El Método Actual define un Gestor en su código

Si el método actual define un gestor para la excepción, este se encontrará en la pila de contextos.

Se busca en la pila de contextos del objeto hasta encontrar el manejador de la excepción, desechándose todas las llamadas a métodos locales que se hayan hecho entre la instalación del `handler` y el momento de producirse la excepción.

Cada método que deba abandonarse por causa de una excepción, libera previamente los objetos locales declarados en la sección Instances del método (se finaliza el método como con Exit).

De esta forma, cuando se ejecute una instrucción `throw`, la ejecución del programa continuará en la dirección del último `handler` ejecutado, ejecutándose sus instrucciones.

5.3.2 El Método Actual no define un Gestor en su código

Puede suceder, sin embargo, que el método en ejecución no defina un manejador para la excepción que se produzca.

En este caso, se retrocede en la secuencia de llamadas que llevó hasta la situación actual, intentando encontrar algún método en la cadena de llamadas, que hubiese declarado un Handler. De esta forma, la excepción se propaga en la cadena de llamadas.

Para resolver el problema de la propagación de excepciones en entornos concurrentes y, posiblemente, distribuidos, la unificación de criterios es la vía más prometedora. Para ello, el área de gestión de excepciones de la máquina se ha definido de forma que se mantenga la analogía entre retorno de un método con o sin excepción.

Retorno en la cadena de invocación a método

Las excepciones se consideran resultados, anormales, pero resultados al fin y al cabo. Por tanto, se gestiona la propagación de excepciones como si fuese el retorno de resultados, cosa que realmente puede considerarse como tal de forma similar a como se realiza en Hybrid[KNP88].

Cuando un objeto termina de servir una invocación a un método, la máquina se encarga de hacerle llegar el resultado al objeto invocador, sea este cual sea, instanciando el objeto resultado o excepción de forma adecuada.

En caso de que sea una excepción, cuando se retorne al contexto del objeto que invocó el método en que se produjo la excepción, la máquina abstracta copia en la referencia exc el objeto excepción que se instancia cuando se lanza la excepción.

Cuando se reanude la invocación del método que recibe la excepción, se determina si el resultado es del tipo esperado o es una excepción. Concretamente, cuando se reanuda un hilo que estaba esperando en una invocación síncrona o una sincronización, la máquina antes de reanudarlo, mira el tipo del resultado.

Si el retorno es normal, se continua normalmente. Si es de tipo excepción, se gestiona como una excepción. Para ello, la máquina abstracta relanza la excepción en el contexto actual y se procede de la misma forma.

Analogía Retorno de Invocación a Método, Retorno Excepción

Se puede ver la analogía existente entre la pareja Throw/handler con Call-Send / Exit, lo que apoya la tesis de gestionar las excepciones como retornos de métodos, estableciendo una correspondencia fuerte entre la forma de devolver resultados de la invocación de un método y la propagación de excepciones no gestionadas.

- La llamada a un método le deja al Exit la dirección donde tiene que retornar, de la misma manera que el Handler se la deja al Throw.
- El Exit retorna la última llamada a un método realizada, lo mismo que el Throw al último Handler ejecutado.
- El Exit descarta todos los Handlers que se encuentren entre él y su llamada al procedimiento. Igualmente el Throw descarta todas las direcciones de retorno de las llamadas a procedimientos que encuentre entre él y su Handler (para ello liberando los objetos de la cláusula Instances).

Los tres puntos anteriores, sugieren que ambos comparten una misma pila en la que se insertan tanto direcciones de retorno como manejadores. Dependiendo de que se ejecute un Exit o un Throw éstos irán retirando elementos de la pila hasta que encuentren su pareja.

5.4 Gestión de Excepciones e Invocación de Métodos Síncrona

En caso de invocación de métodos síncrono, la secuencia de invocación de métodos en distintos objetos se representa como una secuencia de contextos en distintos objetos, de forma similar a un hilo de control, en una aproximación que recuerda a los procesos de UNIX[Bac86], o a los hilos de control en sistemas más modernos.

La invocación de métodos supone la creación de contextos en distintos objetos que forman una secuencia de pilas de contexto que registran la secuencia de llamadas.

El retorno de resultados es muy simple, al tratarse simplemente de desapilar capas o retornar el control al contexto anterior en la pila.

La gestión de excepciones también se produce de forma natural. Durante la ejecución de un método, un objeto puede instalar gestores para una o varias excepciones.

Si se produce una excepción, basta desapilar contextos hasta encontrar un gestor de la excepción, desechando cualquier computación realizada entre la cima de la pila donde se produjo la excepción y la capa de la pila donde se encontró el manejador para la misma.

Esto incluye retornar por la secuencia de hilos creados en distintos objetos, explorando los distintos contextos de ejecución, de tal forma que, si no tiene instalado el manejador en el contexto adecuado, retorna el control, con la excepción, al contexto que realizó la llamada, donde se lanza nuevamente la excepción y se procede de igual forma.

Con esto se van desechando también los contextos de varios objetos por los que ha pasado la ejecución, y que, al no haber instalado ellos mismos un gestor para la excepción, no tienen capacidad para manejarla.

5.5 Gestión de Excepciones e Invocación de Métodos Wait-By-Necessity

La concurrencia entre objetos introducida por la invocación de métodos siguiendo un modelo Wait-by-Necessity, en el que origen y destino prosiguen su ejecución en paralelo, provoca un cambio sustancial en la representación de la computación frente al modelo síncrono.

Al no existir un único hilo de ejecución, ya no es posible seguir una aproximación tipo UNIX, mediante la apilación/desapilación de capas de contexto para representar las sucesivas invocaciones a métodos.

Sin embargo, el modelo de excepciones sigue siendo un componente fundamental ya que permite a la máquina informar a los objetos de la aplicación acerca de sucesos anormales o erróneos.

Cuando se envía un mensaje a un objeto, se retorna inmediatamente el control al objeto origen.

Internamente, esta invocación se representa mediante un objeto virtual que representará el objeto resultado hasta que el método invocado termine e instancie el resultado.

Si durante la ejecución del método llamado se produce un error que no se gestiona en el propio método, el objeto virtual pasa a estar en un **contexto envenenado** y se almacena una referencia a un objeto excepción del tipo que se produjo.

El envío de un mensaje a un objeto en un contexto envenenado, provoca que la llamada retorne inmediatamente una excepción y la máquina abstracta asocia a la referencia rr del método que realiza la llamada, el objeto excepción, continuándose el procesamiento como antes a partir de este punto.

De esta forma se propaga el efecto del error a través de toda la aplicación.

En lenguajes que utilizan este modelo de invocación también se ha estudiado este problema a fondo. Así, por ejemplo, el sistema Merlin, cuyo lenguaje Self utiliza la invocación mediante wait-by-necessity.

5.6 Ventaja de la incorporación de excepciones

El tratamiento de excepciones, como parte aceptada de los lenguajes orientados a objeto y de las metodologías, está incluido en la máquina abstracta. Las ventajas que se podrían señalar a la incorporación de las excepciones son:

Facilitar la tarea de depuración y mantenimiento de las aplicaciones

Los micros tradicionales ofrecen pocos medios, por no decir ninguno, para comprobar si el uso que se está haciendo de las instrucciones es correcto.

Se pueden citar muchos ejemplos. Desde la instrucción `RET` que saca de la pila la dirección de retorno, suponiendo que lo que saca es efectivamente una dirección, hasta el uso de punteros, con una instrucción, `MOVE`, que no puede comprobar si va a escribir en una dirección válida, si se está saliendo del rango, si esa zona de memoria ya se ha liberado, etc… Simplemente se deja que el programador se haga responsable de las consecuencias.

En la máquina abstracta, se pretende que las instrucciones puedan detectar el máximo número posible de situaciones en las que se las esté utilizando incorrectamente. Así, por ejemplo, la instrucción **`delete`** produce una excepción si se le pasa una referencia a todo aquello que no sea un objeto válido (una referencia NULL, un objeto ya liberado, un objeto que es aggregate de otro, etc…).

Lo mismo ocurre con el resto de las instrucciones, lo que supone una inestimable ayuda a la hora de la detección de errores.

Esto es posible gracias a la información que se guarda en las áreas de clases, referencias e instancias. Si se eliminase el uso de excepciones por parte del micro, simplemente en el ejemplo anterior del `delete` se perdería una valiosa oportunidad para detectar algunos errores que serían difícilmente detectables de otra forma.

Buena Costumbre: Programación de Usuario más robusta

Al estar las excepciones tan arraigadas en la forma de programar la máquina, promueve su utilización en los programas de usuario y, por tanto, la generación de un código mas robusto por parte de éstos.

6 Control de la Concurrencia

En un modelo de objetos activo, un objeto puede tener varios hilos activos simultáneamente para dar servicio a varios métodos, iguales o distintos. Es posible incluso, que un mismo objeto pueda tener activo más de un hilo simultáneamente para dar servicio a varios de sus métodos.

Por todo ello, es necesario establecer políticas de sincronización en la ejecución de los métodos de los objetos. Aunque gran parte de estas políticas se implantarán, como

es lógico, a nivel del sistema operativo, es necesario que la máquina abstracta proporcione los mecanismos básicos para poder implantarlas sobre ellos.

De esta forma, gracias a la máquina, y con la colaboración del sistema operativo, se asegurará la consistencia en el estado interno de los objetos. A este filtro que establece la posibilidad de ejecución de los métodos, se le denomina **control de la concurrencia**.

6.1 Soporte de la Máquina Abstracta para el Control de la Concurrencia Entre Objetos

Para sincronizar la ejecución de métodos en distintos objetos, la máquina abstracta proporciona, básicamente, dos herramientas, el paso de mensajes y un objeto básico de la máquina como son los objetos tipo cerradura.

6.1.1 Paso de mensajes

El paso de mensajes propuesto, sea utilizando la semántica síncrona o la wait-by-necessity, obliga a esperar en determinados instantes de tiempo, provocando la sincronización entre la ejecución del método que realiza la invocación y la del que la recibe.

En el caso del **paso de mensajes síncrono**, la sincronización se produce desde el momento en que el hilo en ejecución en el objeto origen, cede el control de la máquina al objeto destino de la invocación, en un símil del mecanismo de **hand-off** de Mach[Bla90], y no lo recupera hasta que, una vez terminada la invocación, la máquina retorna el resultado de la llamada o transmite una excepción hacia atrás en la cadena de llamada-respuesta.

En el caso de **wait-by-necessity**, el hilo en ejecución del objeto origen de la llamada no cede el control sino que, en una aproximación asíncrona, sigue su ejecución. La sincronización se produce cuando el objeto origen o cualquier otro, intenta utilizar un resultado que todavía no está totalmente calculado, es decir, el objeto resultado no está instanciado.

En ese caso, la máquina bloquea al hilo que intenta utilizar el objeto no instanciado hasta que éste último tome el valor correspondiente, momento en el cual, la máquina despierta a todos los hilos esperando en el objeto.

La utilización de un paso de mensajes wait-by-necessity supone que, un objeto que difunda una referencia no instanciada a otros objetos con el fin de que la utilicen, está propiciando la **sincronización de más de dos hilos** lo que es realmente complicado en un paso de mensajes síncrono o asíncrono tradicional.

6.1.2 Objetos tipo cerradura

Para aquellas aplicaciones que necesiten un grano más fino de sincronización, se proporcionan objetos **cerradura**[Ben82] con la semántica tradicional. Son objetos básicos de la máquina, fundamentalmente por razones de eficiencia.

Estos objetos son instancias de la **clase Lock** que forma parte de la jerarquía de **clases primitivas de la máquina**, como se puede ver en el anexo B.

En definitiva, la máquina abstracta ofrece el soporte básico para la concurrencia a través del modelo de comunicación entre objetos y gracias a los mecanismos de control que establece.

6.2 Soporte de la Máquina Abstracta para el Control de la Concurrencia Interna de los objetos

En el control de la concurrencia dentro de los objetos, juega un papel fundamental el Sistema Operativo (Ver Capítulo XVI), que coopera con la máquina mediante el mecanismo de la reflectividad.

Al proporcionar la máquina abstracta la posibilidad de ejecutar en paralelo múltiples métodos de un objeto simultáneamente, creando para ello los hilos pertinentes en la ejecución de la instrucción de paso de mensajes, es necesario establecer un mecanismo de control flexible que permita asegurar un orden en la ejecución de los métodos.

La sincronización dentro de los objetos se produce en la frontera del objeto, en el momento en que recibe una invocación.

Esta invocación será servida, es decir, dará como resultado la creación de un hilo planificable, siempre y cuando un meta-objeto del entorno del objeto lo crea conveniente.

En este caso, la máquina debe tener implementado un mecanismo que permita ceder el control a otro objeto antes de crear el hilo.

Secuencialización de los Métodos

La máquina proporciona a todos los objetos de un mecanismo de control de la concurrencia por defecto.

Este mecanismo dota al objeto de una maquinaria interna de ejecución de métodos totalmente secuencial, donde cada método se ejecuta completamente antes de comenzar la ejecución de cualquier otro método invocado en el objeto.

De esta forma se garantiza la integridad en el **acceso al estado interno del objeto**. Sin embargo, es una política muy **restrictiva** e **inflexible**.

6.2.1 Cooperación con el Sistema Operativo

Es preferible, sin embargo, un mecanismo que permita flexibilidad a la hora de determinar la posible ejecución correcta de varios métodos simultáneamente. Es evidente que la máquina abstracta sola no puede implantar ese mecanismo. Depende de la aplicación.

Máquina abstracta: Mecanismos Básicos de Ejecución

Por tanto, se opta por limitar el papel de la máquina abstracta a proporcionar los mecanismos para ejecutar métodos – el paso de mensajes – y **abrir el diseño** de la misma para que, gracias a la cooperación con el Sistema Operativo, la aplicación pueda personalizar el entorno de ejecución de sus objetos.

La máquina abstracta ofrece los mecanismos básicos de ejecución – **Mecanismos** – y el sistema operativo amplía la máquina abstracta asociando a cada objeto del nivel

base un meta-espacio compuesto por un conjunto de objetos, meta-objetos, que determinan la política de ejecución de métodos – **Política**.

6.2.2 Arquitectura Reflectiva: Cesión de Control al Sistema Operativo

Máquina abstracta y Sistema Operativo, juntos, cooperando gracias a la reflectividad, forman el entorno de computación concurrente para los objetos.

La máquina abstracta, como se vio en el Capítulo XIV, expone ciertas características de su arquitectura interna. Entre estas se encuentra el **control de la concurrencia interna de los objetos**, o modelo de objetos activo. Es decir, el modelo de objetos para la concurrencia interna – objeto serie, objeto totalmente concurrente, etc. – se define en el sistema operativo.

La máquina abstracta cede el control al objeto u objetos del sistema operativo encargados de definir el modelo de objetos para la concurrencia, para que lleven a cabo el control del entorno de ejecución de los objetos.

La máquina abstracta cede el control a este objeto del sistema operativo lo que se consigue mediante la invocación de métodos en el objeto del sistema operativo correspondiente – reflectividad explícita – y también mediante la cesión implícita de control por parte de la máquina a objetos del sistema operativo (modificación del flujo de control al método concreto del meta-objeto correspondiente) ante algunos eventos como invocación a métodos – reflectividad implícita.

Circunstancias en las que se cede el control al meta-objeto

Existen puntos en la ejecución de la máquina que provocan que esta ceda el control al objeto del sistema operativo encargado del control de la concurrencia en el espacio base.

Todas estas circunstancias modifican de alguna forma el entorno de ejecución del objeto del nivel base.

- Invocación de un método. La invocación de un método provoca un intento de ejecutar un método en un objeto, lo que hace necesario chequear la conveniencia de tal ejecución en el estado actual del objeto.
- Suspensión de la ejecución de un método. La suspensión de un método puede provocar que la ejecución de otros métodos pase a ser conveniente.
- Reanudación de la ejecución de un método. Contrariamente a lo anterior, la reanudación de un método, suele provocar que algún otro deje de ser ejecutable.
- Finalización de la ejecución de un método. Igual que en el caso de la suspensión, la finalización de un método puede provocar que otros pasen a ser ejecutables.

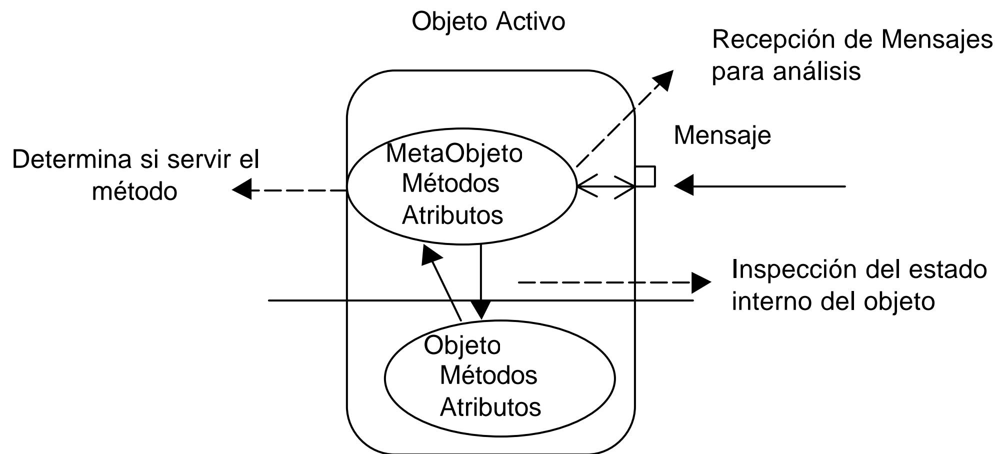


Figura 15.7 Objetos Activos que analizan los métodos y determinan si servir o demorar su ejecución

Esta invocación será servida, es decir, dará como resultado la creación de un hilo planificable, siempre y cuando el sistema operativo – o lo que es lo mismo el metaespacio que configura el entorno del objeto – lo crea conveniente.

Aumento de la Semántica del Objeto

El metaespacio de un objeto amplía considerablemente la semántica del mismo. Los objetos reciben mensajes (invocaciones de métodos potenciales), que se almacenan y que el meta-espacio – que se ejecuta virtualmente en un hilo dedicado – se encarga de gestionar de alguna forma que no necesariamente tiene por qué ser serie.

La adición de concurrencia dentro del objeto se logra añadiendo al metaespacio la capacidad de dar de paso varios mensajes simultáneamente, lo que virtualmente provocaría la creación de nuevos hilos o contextos que sirviesen métodos específicos del objeto, aunque la creación de tales hilos la realizaría físicamente, la propia máquina abstracta.

Aunque la política básica implementada es que sólo un hilo de control puede estar activo dentro de un objeto, el modelo es suficientemente general para permitir que varios estén ejecutándose simultáneamente en caso de multiprocesamiento.

7 Colaboración de la máquina abstracta en el mecanismo de Planificación

La planificación se refiere a la capacidad del sistema de poner la máquina al servicio de distintos hilos activos de cualquiera de los objetos en distintos momentos de tiempo. Necesita una **política** que determine cuándo y por quién se realiza el cambio, y un **mecanismo**, que realice el cambio propiamente dicho.

Como en el caso de los sistemas operativos tradicionales, también aquí se propugna una división de la política y el mecanismo encargándose la máquina de llevar a cabo las acciones englobadas bajo el nombre de mecanismo que cambian un hilo por otro, de entre los distintos objetos activos en el sistema y, el sistema operativo se encargará de

realizar las tareas propias de todo planificador eligiendo, de entre todos los hilos elegibles, el más adecuado, según la política que se considere más conveniente.

7.1 Ciclo de Ejecución de la Máquina Abstracta

La máquina abstracta tiene un ciclo de ejecución muy sencillo basado en la ejecución de instrucciones, dentro del conjunto de instrucciones que forma su lenguaje, siguiendo la pauta que le marca el Sistema Operativo.

El ciclo de ejecución de la máquina consiste en ejecutar la instrucción señalada por el valor de PC del hilo actual, aquel que se encuentra referenciado en la referencia del sistema **ct**, **current thread**, y actualizar el valor de PC en el hilo actual.

Este ciclo de ejecución continúa indefinidamente y sólo se ve interrumpido en el caso de que el hilo actual libere voluntariamente la máquina o la requisen, en cuyo caso, se cambia el valor de la referencia ct, para que la máquina siga el ciclo de ejecución con otro hilo.

Liberación voluntaria

La liberación voluntaria de la máquina abstracta se produce en caso de que el hilo en ejecución termine o se suspenda por alguna razón.

- **Finalización**. La máquina abstracta procesa la finalización de un hilo de ejecución o finalización de la ejecución de un método, como el final de una invocación a método, tal y como se ha descrito en el apartado Retorno de Resultados.
- **Suspensión**. En caso de que el hilo se suspenda por cualquier razón, lo que sucede cuando, desde algún método se invoca el método **suspend** del objeto hilo, cuando se bloquea el hilo al intentar adquirir una cerradura, cuando realiza una invocación síncrona, mientras esta no termine, o si es asíncrona y el objeto resultado todavía no está instanciado, la máquina abstracta, tras salvar el contexto de ejecución en el hilo, para poder reanudar su ejecución posteriormente, invoca al método adecuado del planificador con el fin de realizar un cambio de contexto[TW98].

Requisamiento

La máquina abstracta también permite el requisamiento[TW98] de hilos, es decir, ofrece mecanismos para permitir quitarle la máquina a un hilo en ejecución que no la ceda de forma voluntaria.

El requisamiento suele ser debido a la implantación de una política de planificación basada en prioridades o en intervalos de tiempo.

Para dar soporte al primer caso, la máquina abstracta implementa interrupciones temporales que se verá con posterioridad. En el segundo caso, la máquina abstracta, cuando crea un nuevo hilo, inspecciona la política de planificación y, en caso de que esté basada en prioridades, invoca el método adecuado en el planificador, con el fin de mantener la política de planificación.

7.2 Interrupciones de Reloj

La máquina abstracta implementa un mecanismo de interrupciones temporales típico, que emite interrupciones cada cierto número de ticks de reloj.

La gestión de esta interrupción, llevada a cabo por la propia máquina abstracta, consistirá en ejecutar las acciones necesarias para gestionar el final de un intervalo de tiempo.

Esto supone **salvar el contexto** del hilo en ejecución e **invocar el método correspondiente en el objeto planificador**.

7.3 Cambio de Contexto

El **cambio de un contexto de ejecución** por otro se produce, habitualmente, cuando el hilo actual se bloquea por alguna razón, termina o lo interrumpen, normalmente, una interrupción de reloj.

En caso de que sea necesario un cambio de hilo de ejecución por otro en la máquina abstracta, esta es una operación sencilla que, virtualmente sólo requiere **salvar el contexto actual**, si no se había salvado previamente, y **cargar el nuevo contexto**.

Salvar Contexto

Dada la propia arquitectura de la máquina abstracta, carente de registros, las operaciones se realizan sobre el propio objeto interno hilo que representa la ejecución del método. Por ello, la operación de salvar contexto se limita a actualizar las variables locales de los hilos que se estén modificando.

Restaurar Contexto

Cargar un nuevo contexto implica el cambio de una referencia (referencia al hilo de ejecución activo) en el área de referencias del sistema, **ct**.

A partir de ese momento, la máquina tomará como área de trabajo (referencias a parámetros, valor que determina la siguiente instrucción a ejecutar, ...) la que determine el hilo referenciado en ct.

Evidentemente, la elección del hilo nuevo queda fuera del ámbito de la operación de cambio de contexto y pertenece al conjunto de responsabilidades del planificador.

7.4 Planificador del Sistema.

El sistema operativo, mediante una política de planificación, decide cuál de los hilos activos en los objetos, es el más adecuado para ejecutarse y le comunica tal información a la máquina abstracta, señalándole la referencia del contexto elegido.

Para ello, en el área de referencias del sistema, se almacena la referencia ct, current thread, en el que se señala la referencia del contexto a ejecutar.

7.4.1 El planificador de la máquina

La política de planificación definida por el Sistema Operativo (Ver capítulo XVI), estará determinada por un objeto planificador que realice las tareas propias de su género. Dado que se trata de un Sistema Operativo Orientado a Objeto, los planificadores serán a su vez objetos, que deberán ser planificadores para ejecutar su política de planificación y elegir el mejor contexto para ejecutar.

La recursión finaliza con un objeto planificador raíz de toda la jerarquía de planificadores, instanciado cuando se crea la máquina abstracta de forma similar al MetaCore de Apertos [Yok92].

La referencia root-sched (scheduler o planificador raíz de la jerarquía de planificación) a dicho objeto, se contará entre las referencias al sistema, y tendrá el control de la máquina abstracta según sea el tipo de política implementada.

Lo habitual será que se implemente un planificador RR, con lo que conseguirá el control de la máquina cada vez que el quantum de tiempo asignado a un objeto expire.

Por otra parte, además del planificador raíz, lo habitual será que existan otros planificadores, instanciados por las aplicaciones, que reciban el control de la máquina del planificador raíz y lo cedan a su vez a los hilos de tal aplicación o a otros planificadores.

En cada momento, hay un planificador activo en el sistema. La referencia del sistema sched se encarga de señalar cuál de los planificadores en el sistema es.

7.4.2 Planificación a bajo nivel

Se verá en el siguiente capítulo cómo la construcción de una jerarquía de planificadores en tiempo de ejecución puede beneficiar la ejecución de las aplicaciones actualmente en el sistema.

En el nivel más bajo de esta jerarquía de planificadores, un objeto planificador recibirá tiempo de su planificador de rango superior y le devolverá el control cuando termine el tiempo o bien termine su trabajo.

En el intermedio de ambos eventos, decide cuál de los hilos activos en los objetos, es el más adecuado para ejecutarse ahora y le comunica tal información a la máquina abstracta, señalándole la referencia del hilo elegido.

Para ello, dado que los objetos hilos son objetos internos, que carecen de referencias, es necesario exponer la ejecución de los objetos (Ver apartado 8), lo que hará que se puedan representar los hilos como objetos en la máquina y no como objetos internos para su manejo en determinadas operaciones.

Así, simplemente invocando el método run en el hilo escogido, se transmitirá a la máquina abstracta el mensaje de que comience o continúe la ejecución de ese método.

Para ello, en la referencia del sistema ct, current thread, se almacena la referencia al hilo actual, señalando así el contexto a ejecutar.

A partir de este momento, la máquina abstracta comienza a ejecutar las instrucciones del método señalado, una tras otra, modificando, en cada caso, el contador en el contexto de ejecución y pasando a ejecutar la siguiente instrucción.

8 Arquitectura Reflectiva: Adecuación de la máquina abstracta para permitir su extensión por el Sistema Operativo

Para poder lograr la modificación de la máquina abstracta en tiempo de ejecución y permitir de esta forma la cooperación entre máquina abstracta y Sistema operativo, es

preciso modificar en algunos aspectos la arquitectura interna de la máquina, dotándola, inicialmente, de **Reflectividad Estructural**.

Se trata, fundamentalmente, de convertir los objetos del sistema en objetos visibles y manipulables por el usuario. Para ello se definen, en el lenguaje definido por la máquina abstracta, un conjunto de clases básicas, que forman el módulo de Reflectividad Estructural de la máquina.

Esta jerarquía de clases básicas cosifican o reflejen, aspectos de implantación de la arquitectura de la propia máquina. A continuación se verán los aspectos a exponer y las razones por las que es necesario exponer tales aspectos.

A partir de estas clases, la información elegida se mantendrá en el meta-nivel del objeto concreto, representada como objetos de primera clase en tiempo de ejecución, con una serie de métodos que podrán ser invocados dinámicamente, aunque siempre de forma síncrona, y que provocarán la modificación de dichos elementos en tiempo de ejecución, según las necesidades de la aplicación.

La gestión y manipulación de estos objetos se lleva a cabo por distintos meta-objetos que componen el entorno del objeto base, pudiendo aplicarse sobre ellos la característica de **introspección** (poder saber qué, cómo y por dónde se está ejecutando, qué está siendo retrasado su ejecución, etc.) e **intercesión**, al poder modificarse dichos métodos. Esta última propiedad es fundamental para la consecución de la **Reflectividad del Comportamiento** o adecuación del entorno dinámicamente.

8.1 Exponer elementos de la arquitectura de la máquina.

Un primer elemento a exponer es la propia arquitectura de la máquina. Para ello, se exponen las áreas constituyentes de la arquitectura que, a partir de este momento, estarán representadas por objetos que existen en tiempo de ejecución.

Con ello no se pretende modificar su comportamiento, es decir, no se pretende modificar ni la funcionalidad del área ni el modo en que la lleva a cabo, aunque se verá que, a partir de la arquitectura reflectiva de la que se dota a la máquina esto sería posible, en ciertos aspectos.

Se trata más bien, de hacer accesible esa funcionalidad al exterior, permitiendo así que los objetos puedan acceder a la funcionalidad interna de la máquina en tiempo de ejecución de la misma forma que si accediesen a otro objeto cualquiera, mediante la invocación de un método.

8.1.1 Área de Clases

La exposición del área de clases de la máquina como un objeto, tiene como finalidad permitir el acceso y modificación del área de clases de la máquina.

Gracias a este objeto, es posible averiguar qué clases contiene el área de clases, es decir, qué clases hay definidas en un momento determinado (Introspección), así como hacer visibles y manipulables tales clases gracias a la exposición del área de clases y de las propias clases.

Es el primer paso para poder crear clases de usuario en tiempo de ejecución y con ello modificar la jerarquía de clases dinámicamente. Esto, junto con la exposición de

instancias y referencias, es fundamental para la consecución de la Reflectividad del Comportamiento.

8.1.2 Área de Instancias

La exposición del área de instancias de la máquina como un objeto, tiene como finalidad permitir el acceso y modificación del área de instancias de la máquina.

Igual que en el caso anterior, el objeto que representa al área de instancias en tiempo de ejecución, permite conocer qué instancias hay creadas en tiempo de ejecución(Introspección), gracias a lo cual, junto con la exposición de las propias instancias, los propios objetos son visibles y manipulables en tiempo de ejecución.

Por otro lado, permite también crear instancias de una clase en tiempo de ejecución.

8.1.3 Área de Referencias

El área de Referencias no existe como tal en la máquina abstracta, es un área virtual que sirve, únicamente para establecer la correspondencia entre la referencia a un objeto y su identificador dentro de la máquina.

Sin embargo, para cumplir un papel similar se exponen las Referencias de los objetos.

Esto permite el acceso y creación de nuevas referencias en la máquina abstracta en tiempo de ejecución. La creación de nuevas referencias en tiempo de ejecución es fundamental para poder definir clases en tiempo de ejecución: los agregados, asociados, ancestros de una clases, argumentos de un método, etc.

Permite conocer el nombre de una referencia, la clase a la que pertenece o si está libre o instanciada.

8.1.4 Área de hilos.

Por último, la exposición del área de hilos del sistema permite el acceso al área de hilos de la máquina abstracta.

Esto permite conocer los hilos que hay en el sistema, tanto el número de ellos como obtener los hilos propiamente dichos.

Esto permite poder implantar políticas de balance de carga en Sistemas Distribuidos, gracias a la posibilidad de conocer, en tiempo de ejecución, la carga de cada máquina.

8.2 Exponer elementos estructurales de la implantación en tiempo de ejecución de los objetos

Además de la exposición de la arquitectura interna de la máquina, es necesario también exponer la composición de las instancias en tiempo de ejecución, es decir, la implantación de los objetos en tiempo de ejecución, de forma que no sólo se pueda acceder a la información de los mismos en tiempo de ejecución (Introspección), sino que incluso, se puedan manipular las instancias dinámicamente.

De la misma forma que antes, para lograrlo, se exponen los distintos elementos que constituyen las instancias y que, a partir de este momento, podrán ser creados y manipulados en tiempo de ejecución.

8.2.1 Las Clases

La exposición de las clases tiene una doble finalidad. En primer lugar, permitir el acceso a las clases en tiempo de ejecución así como poder realizar introspección sobre las mismas, conociendo su composición, sus métodos, la cadena de herencias, los agregados y asociados, etc.

Por otro lado, permitir la creación de clases en tiempo de ejecución, definiendo completamente sus métodos, agregados, etc.

Esto, junto con la exposición del área de clases permitirá añadir nuevas clases al área de clases en tiempo de ejecución, ampliando la jerarquía de clases de forma dinámica.

8.2.2 Las instancias

La exposición de las instancias, es decir, de los objetos en tiempo de ejecución, permite el acceso a las instancias de la máquina abstracta.

Gracias a la exposición de las instancias, la máquina permite conocer la composición de las instancias que hay creadas, permitiendo el acceso a sus agregados y asociados, la clase a la que pertenece, etc.

8.2.3 Los métodos

Si la exposición del área de clases permitía obtener los métodos que componen la clase, la exposición de los métodos en sí, permite el acceso y modificación de los mismos en tiempo de ejecución.

De esta forma se permite, no sólo crear nuevos métodos en tiempo de ejecución, imprescindible para definir nuevas clases dinámicamente, sino también realizar introspección sobre los métodos, conociendo sus argumentos, variables locales, instrucciones que lo componen, tipo de retorno, etc.

8.2.4 Las Instrucciones

Esta clase, que exponen las instrucciones de la máquina, se crea fundamentalmente, para permitir la definición de clases en tiempo de ejecución, permitiendo dotar de instrucciones a sus métodos.

Fundamentalmente, permite crear nuevas instrucciones, de entre el conjunto de instrucciones de la máquina, en tiempo de ejecución que luego serán asignadas a algún método.

También permite ciertas capacidades de introspección, al permitir averiguar características sobre las instrucciones como de qué instrucción se trata.

8.3 Exponer la Engine en tiempo de ejecución

Aunque podría estar englobado en el apartado anterior, se expone el entorno en tiempo de ejecución de los objetos, permitiendo averiguar y actuar sobre los métodos que se están ejecutando, el estado de cada ejecución, etc.

Esta información se mantiene en el meta-nivel del objeto concreto y es gestionada por distintos meta-objetos que componen el entorno del objeto base, como el planificador.

De esta forma, se puede aplicar sobre ella la característica de **introspección** (poder saber qué, cómo y por dónde se está ejecutando, qué está siendo retrasado su ejecución, etc) e incluso **intercesión**, al poder modificarse algunas características dinámicamente, como la prioridad, el modo de ejecución y otros.

8.3.1 Área de Ejecución del Objeto

La representación en tiempo de ejecución de un objeto comprende no sólo los elementos estructurales vistos antes, sino también los elementos de ejecución que representan los métodos que se están ejecutando en el objeto.

Aunque es un elemento de implantación de la máquina se expone al exterior con dos finalidades fundamentales.

En primer lugar, poder averiguar qué métodos del objeto se están ejecutando y el estado de su ejecución.

Y, fundamentalmente, para poder implantar la reflectividad del comportamiento. Esta última consistirá en modificar partes de la máquina mediante objetos del sistema operativo. Para ello, uno de los elementos fundamentales en tiempo de ejecución a los que hay que acceder es la computación. Gracias a la exposición del área de threads de la máquina y del conjunto de hilos de cada objeto, es posible realizar acciones como la planificación a más alto nivel.

8.3.2 Hilos de Ejecución

La exposición del área de ejecución de un objeto expone al exterior los distintos hilos de ejecución activos en cada objeto. La exposición de los hilos de ejecución no es nueva, otras máquinas como Smalltalk[Riv96] ya proponen algo similar.

La exposición de los hilos permite el acceso y modificación del propio hilo, es decir, de la ejecución de un método.

Ofrece la posibilidad de conocer qué método se está ejecutando, obtener el estado del hilo, las referencias a las instancias locales, así como conocer la cadena de llamadas que han llevado hasta la situación actual.

8.4 Exponer el MetaEspacio

Además de la exposición de la arquitectura interna de la máquina, es necesario también exponer la composición de las instancias en tiempo de ejecución, es decir, la implantación de los objetos en tiempo de ejecución, de forma que no sólo se pueda acceder a la información de los mismos en tiempo de ejecución (**Introspección**), sino que incluso, se puedan manipular las instancias dinámicamente (**Intercesión**).

De la misma forma que antes, para lograrlo, se exponen los distintos elementos que constituyen las instancias y que, a partir de este momento, podrán ser creados y manipulados en tiempo de ejecución.

8.4.1 El MetaEspacio de un Objeto

Un objeto pertenece a un metaespacio en tiempo de ejecución, que señala el comportamiento de la máquina abstracta en determinados aspectos, para ese objeto. Un metaespacio(Ver Capítulo XVI), no es más que un conjunto de objetos, cuyos métodos modifican el comportamiento de la máquina abstracta.

Un objeto puede determinar un comportamiento por defecto de la máquina y, sin embargo, puede desear cambiarla, en función de su estado y su entorno en cualquier momento, en tiempo de ejecución.

La exposición del metaespacio permite el acceso y manipulación del mismo lo que permite dotar a la máquina de un comportamiento distinto en tiempo de ejecución.

Permite interrogar al objeto acerca de los aspectos expuestos – aquellos que tendrán asociado un metaobjeto – y conseguir las referencias a los distintos meta-objetos.

Junto con la exposición de Clases, permite definir nuevos comportamientos – nuevos metaobjetos – en tiempo de ejecución, y, unido a la exposición de instancias y del metaespacio, permite modificar, en tiempo de ejecución, el meta-espacio de un objeto.

8.4.2 Los metaobjetos

Permite el acceso y modificación en tiempo de ejecución de un metaobjeto. De esta forma, se puede interrogar a un metaobjeto acerca del aspecto de la máquina que expone, de sus métodos o lo que es lo mismo, cómo modifica el comportamiento de la máquina, etc.

Igualmente, y gracias a la exposición de métodos e instrucciones, será posible modificar un metaobjeto en tiempo de ejecución, permitiendo de esta forma variar su comportamiento, dinámicamente.

9 Ventajas del uso de una máquina abstracta

Con la utilización de una máquina abstracta que siga la arquitectura de referencia anterior se consiguen una serie de ventajas:

9.1 Portabilidad y heterogeneidad

La utilización de una máquina abstracta dota de portabilidad al sistema. El juego de instrucciones de alto nivel puede ejecutarse en cualquier plataforma donde este disponible la máquina abstracta. Por tanto, los programas escritos para máquina abstracta son portables sin modificación a cualquier plataforma. Basta con desarrollar una versión (emulador o simulador) de la máquina para ejecutar cualquier código de la misma sin modificación: el código es portable y entre plataformas heterogéneas. Como todo el resto del sistema estará escrito para esta máquina abstracta, el sistema integral

completo es portable sin necesidad de recompilar ni adaptar nada (únicamente la máquina abstracta).

9.2 Facilidad de comprensión

La economía de conceptos, con un juego de instrucciones reducido basado únicamente en la OO facilita la comprensión del sistema. Está al alcance de muchos usuarios comprender no sólo los elementos en el desarrollo, si no también la arquitectura de la máquina subyacente. Esto se facilita aún más al usarse los mismos conceptos de la OO que en la metodología de desarrollo OO.

9.3 Facilidad de desarrollo

Al disponer de un nivel de abstracción elevado y un juego de instrucciones reducido, es muy sencillo desarrollar todos los elementos de un sistema integral orientado a objetos, por ejemplo:

9.4 Compiladores de lenguajes

Es muy sencillo desarrollar compiladores de nuevos lenguajes OO o lenguajes ya existentes. La diferencia semántica que debe salvar un compilador entre los conceptos del lenguaje y los de la máquina es muy reducida, pues el alto nivel de abstracción OO de la máquina ya está muy cercano al de los lenguajes. El esfuerzo para desarrollar compiladores se reduce.

9.5 Implementación de la máquina

En lo que concierne a la implementación de la propia máquina, también se obtienen una serie de ventajas.

9.6 Esfuerzo de desarrollo reducido

El juego de instrucciones reducido hace que el desarrollo de un simulador de la máquina abstracta sea muy sencillo. No hay que programar código para un número muy grande de instrucciones, con lo que el tiempo necesario y la probabilidad de errores disminuyen.

9.7 Rapidez de desarrollo

Además, al ser la interfaz de la máquina independiente de estructuras internas, se puede elegir la manera interna de implementarla más conveniente. Para desarrollar rápidamente una implementación se pueden utilizar estructuras internas más sencillas aunque menos eficientes.

9.8 Facilidad de experimentación

Todo ello facilita la experimentación. Se puede desarrollar rápidamente una máquina para una nueva plataforma para hacer funcionar el sistema. Posteriormente, debido a la independencia de la interfaz, se puede experimentar con mejoras internas a la máquina: optimizaciones, nuevas estructuras, etc. sin necesidad de modificaciones en las aplicaciones.

9.9 Buena plataforma de investigación

Todas las ventajas anteriores la constituyen en la base para una buena plataforma de investigación. La facilidad de comprensión y desarrollo, y la portabilidad y heterogeneidad permitirán que más personas puedan acceder al sistema sobre cualquier equipo y utilizarlo como base para la investigación en diferentes áreas de las tecnologías OO: el ejemplo anterior de lenguajes OO, bases de datos, etc. Esto se aplica también a la propia máquina en sí, cuya estructura permite una fácil experimentación con diferentes implementaciones de la misma.

10 Minimización del problema del rendimiento de las máquinas abstractas

Como inconveniente de la utilización de máquinas abstractas se cita comúnmente el escaso rendimiento de las mismas [SS96]. Se manejan cifras que otorgan a los intérpretes una velocidad entre uno y dos órdenes de magnitud más lenta que el código compilado [May87]. Por ejemplo, ciertos intérpretes de Java suelen ejecutar los programas a un 10% de la velocidad de un programa en C equivalente [Way96]. En muchos casos las diferencias muy exageradas son en casos extremos o en comparaciones viciadas de programas OO que por su estructura no se pueden comparar con otros en lenguajes convencionales como C.

En cualquier caso, la propia naturaleza de una máquina abstracta necesita la utilización de un programa para simularla, con lo que el rendimiento tiene que ser menor que si se usase el hardware directamente.

Sin embargo, existen una serie de razones que hacen que este problema del rendimiento no sea tan grave, e incluso llegue a no tener importancia:

10.1 Compromiso entre velocidad y conveniencia aceptado por los usuarios

El simple rendimiento no es el único parámetro que debe ser tenido en cuenta en un sistema. Lo verdaderamente importante es la percepción que tengan los usuarios de la utilidad del sistema, que es función del esfuerzo de programación, la funcionalidad de las aplicaciones, y el rendimiento conseguido. El éxito que ha alcanzado la plataforma Java [KJS96] está basado en la utilización de una máquina abstracta. Esto demuestra que el compromiso entre el rendimiento y la conveniencia de los beneficios derivados del uso de una máquina abstracta ya es aceptado por los usuarios con implementaciones sencillas de una máquina abstracta.

10.2 Mejoras en el rendimiento

Existen una serie de áreas con las que se puede mejorar el rendimiento de las máquinas abstractas, reduciendo aún más el inconveniente de su (aparente) pobre rendimiento.

10.2.1 Mejoras en el hardware

La tendencia exponencial del aumento de rendimiento del hardware, junto con la disminución de su precio se ha utilizado históricamente en la informática para elevar el nivel de abstracción [BPF+97]. Las máquinas abstractas son una continuación de esta tendencia, como lo fue el paso del ensamblador a los lenguajes de alto nivel. La potencia adicional se destina a elevar el nivel de abstracción (utilizar una máquina abstracta), que hace que los proyectos sean más baratos de desarrollar (hay una relación no lineal entre el nivel de abstracción y el coste de un proyecto). Los beneficios de un mayor nivel de abstracción compensan la pérdida de rendimiento.

Por otro lado, si con los procesadores convencionales actuales e implementaciones sencillas de máquinas abstractas se ha conseguido una gran aceptación, el aumento de potencia del hardware no hará más que minimizar aún más el problema aparente del rendimiento.

10.2.2 Optimizaciones en la implementación de las máquinas. Compilación dinámica (justo a tiempo)

La implementación de una máquina abstracta puede optimizarse para que la pérdida de rendimiento sea la menor posible. Una técnica de optimización es la compilación dinámica o justo a tiempo (JIT, *Just In Time*). Se trata de optimizar la interpretación del juego de instrucciones de la máquina. En lugar de interpretarlas una a una, se realiza una compilación a instrucciones nativas (del procesador convencional) del código de los métodos en el momento de acceso inicial de los mismos (justo a tiempo) [ALL+96]. Los siguientes accesos a ese método no son interpretados de manera lenta, si no que acceden directamente al código previamente compilado, sin pérdida de velocidad. Este código nativo compilado puede ir siendo optimizado aún más en cada llamada adicional (generación incremental de código) [HU94].

Ciertas implementaciones de máquinas abstractas que utilizan esta técnica han resultado sólo de 1.7 a 2.4 veces más lentas que un código C++ equivalente optimizado [Höl95].

Otro ejemplo de la posibilidad de optimización en la implementación de máquinas abstractas se comprueba en el producto Virtual PC de la compañía Connectix Corporation. Virtual PC es una aplicación Macintosh que emula un ordenador PC completo por software sobre una plataforma PowerPC-Mac. Se alcanzan relaciones de 3 instrucciones PowerPC por cada instrucción Pentium emulada y de 5 a 9 instrucciones PowerPC por cada 3 instrucciones Pentium [Tro97].

10.2.3 Implementación en hardware

En aquellos casos en que no sea aceptable la pequeña pérdida de rendimiento de una máquina optimizada, se puede recurrir a la implementación de la máquina en hardware.

Esta implementación en hardware especializado ofrecería un rendimiento superior al de cualquier implementación software [Way96].

10.3 Resumen

Las propiedades fundamentales que debe tener una máquina abstracta que de soporte a un SIOO son el modelo único de objetos que implementará, con identificador único de objetos, la uniformidad en la OO, una interfaz de alto nivel con un juego de instrucciones reducido y la flexibilidad. Una estructura de referencia para este tipo de máquina abstracta se compone de cuatro elementos fundamentales: áreas para las clases, instancias, referencias para los objetos; referencias del sistema y jerarquía de clases básicas. El juego de instrucciones permitirá describir las clases (herencia, agregación, asociación y métodos) y el comportamiento de los métodos, con instrucciones de control de flujo y excepciones, creación y borrado de objetos e invocación de métodos.

Las ventajas del uso de una máquina abstracta como esta son básicamente la portabilidad y la heterogeneidad, y la facilidad de comprensión y desarrollo, que la hacen muy adecuada como plataforma de investigación en las tecnologías OO. El inconveniente de la pérdida de rendimiento por el uso de una máquina abstracta se ve contrarrestado por la disposición de los usuarios a aceptar esa pérdida de rendimiento a cambio de los beneficios que ofrece una máquina abstracta. Por otro lado, mejoras en el hardware y optimizaciones en la implementación de las máquinas minimizan aún más este problema.

Capítulo XVI

Sistema Operativo: El Metaespacio

Estructurar el entorno OO como la combinación reflectiva de Sistema Operativo y máquina abstracta, conlleva que el Sistema Operativo realice tareas que, de otro modo, estarían integradas en la máquina. Esto dota al sistema de gran flexibilidad, al permitir sustituir partes del sistema operativo por otras con distinta implantación, cosa que, si tal funcionalidad estuviese codificada internamente en la máquina abstracta, sería imposible.

Determinar qué tareas debería llevar a cabo la máquina y, por tanto serían fijas e inamovibles, y cuáles debería llevar a cabo el Sistema Operativo y, por tanto, serían adaptables al entorno concreto del objeto o la aplicación, supone siempre un compromiso entre eficiencia y flexibilidad.

1 El mundo de Objetos del Sistema Operativo

Los sistemas operativos configurables tradicionales, normalmente, ofrecen un conjunto fijo y limitado de funciones. Sin embargo, con el advenimiento de nuevas generaciones de aplicaciones, donde estas son capaces de cambiar sus requerimientos durante la ejecución, este soporte del sistema operativo se vuelve limitado.

El sistema operativo debe ser capaz de soportar la adaptación en tiempo de ejecución. Esto significa que no sólo ofrezcan funcionalidad tradicional como la planificación, sino una interfaz que permita a las aplicaciones controlar el modo en que el sistema operativo ofrece tal funcionalidad, es decir, la política de planificación.

La solución al problema es emplear una arquitectura reflectiva que facilita la adaptación, donde objetos definidos por las aplicaciones pueden elegir el modo en que se ejecutan en su entorno, eligiendo entre varias posibles configuraciones.

El Sistema Operativo: Objetos como los demás

Los servicios del Sistema Operativo se implantan mediante un conjunto de objetos, no diferenciables en nada de los objetos de usuario, excepto, evidentemente, en la tarea que realizan y, derivado de esto, en los privilegios que se necesitan para invocar sus métodos. Otros sistemas como Apertos[Yok92] o Tigger[ZC96], propugnan el diseño de un sistema operativo como un marco de soporte de objetos, instanciable y configurable en tiempo de ejecución.

Al conjunto de objetos que realizan tareas propias del Sistema Operativo, que pueden ir desde la planificación hasta tareas de seguridad, servidores de nombres, etc. se les denomina **espacio de objetos del sistema**.

De esta forma, las llamadas al sistema tradicionales se transforman en meras invocaciones a métodos, que exigirán al objeto origen de la invocación a estar dotado de determinados permisos sobre el objeto del sistema operativo al que invoca.

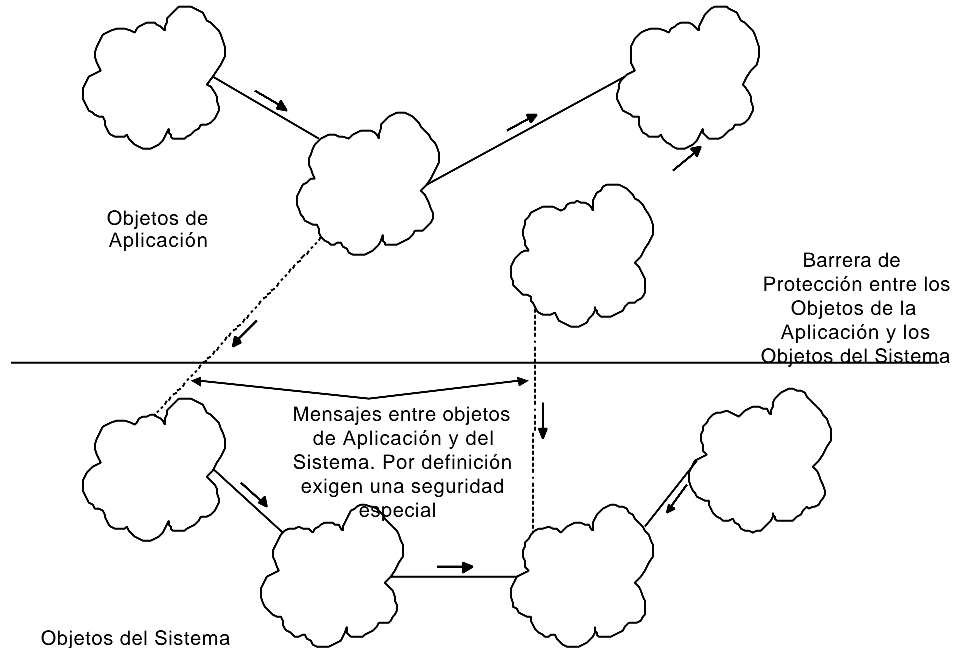


Figura 16.1 Espacio de Objetos del Sistema.

Reflectividad(Máquina, Sistema Operativo)

Se propone aplicar el mecanismo de reflectividad a la organización del sistema de forma que la máquina transfiera el control a los métodos de tales objetos en lugar de llevar a cabo ella misma la tarea, lo que conferiría un carácter altamente inflexible al sistema. De esta forma, los objetos del sistema operativo extienden el comportamiento de la máquina en determinadas áreas.

En concreto, esta tesis se centra en aplicar la reflectividad a la organización de máquina abstracta y sistema operativo para **dotar de un entorno concurrente a los objetos**. Se propone que el Sistema Operativo extienda la máquina en áreas como la **Comunicación** entre objetos, maximizando la concurrencia entre objetos, **el modelo de objetos** ofrecido por la máquina, proporcionando a los objetos mecanismos necesarios para el Control de la concurrencia interna de los mismos, y la **Planificación**.

Meta-Definición del Entorno

Conceptualmente, el meta-nivel de un objeto no tiene por qué ser un único meta-objeto, sino más bien una meta-definición o grupo de objetos, cada uno de los cuales define una faceta particular del comportamiento del objeto. A continuación se identifican cuatro meta-definiciones que se corresponden con las áreas de interés alrededor de las que gira esta tesis, ya presentadas en capítulos anteriores: Descripción del Envío de Mensajes, Descripción de la Recepción de Mensajes, Descripción de la Ejecución de los Métodos y Descripción de la Planificación de Actividades.

Puede haber más o menos componentes dependiendo de las necesidades particulares del sistema. Independientemente de esto, cada uno proporciona la definición de una parte del comportamiento requerido por el objeto base.

El MOP es la unión de los protocolos definidos por todos los posibles componentes.

Comportamiento del Objeto

El comportamiento del objeto se ve modificado asociando explícitamente meta-componentes con un objeto. Las operaciones definidas por estos meta-componentes sobreescriben las especificaciones contenidas en el metaobjeto por defecto (la máquina abstracta). Cualquier objeto puede ser un meta-objeto siempre y cuando responda al protocolo requerido para el papel que juega. A continuación se muestra un ejemplo de estructura de meta-componentes.

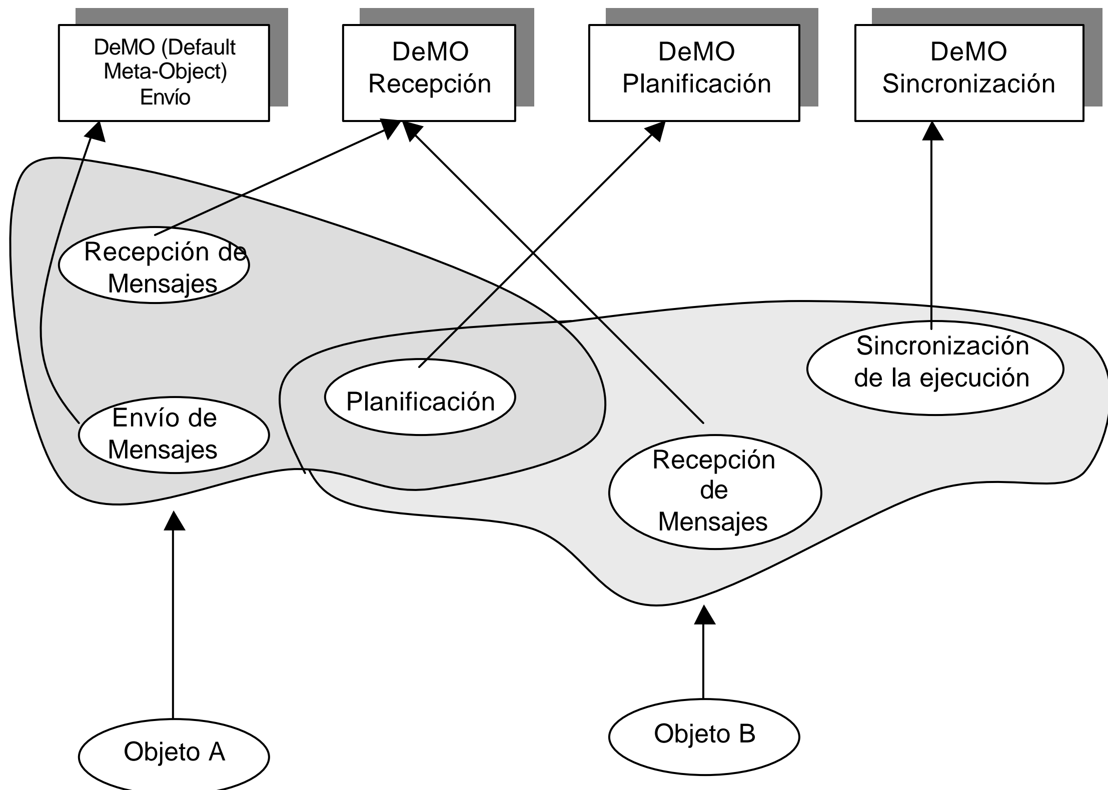


Figura 16.2 Ejemplo de estructura de meta componentes

Procesamiento Reflectivo

El procesamiento reflectivo o procesamiento en el meta-nivel, se lleva a cabo mediante la comunicación directa con el meta-componente(s) deseado(s) o bien de forma implícita pasándole la máquina el control directamente. Los meta-componentes son igual que cualquier otro objeto en el sistema – solo que su dominio es la descripción del comportamiento de un objeto. Como resultado, la distinción entre niveles está prácticamente desdibujada.

2 El Sistema de Comunicación del Sistema Operativo

La invocación de métodos o paso de mensajes entre objetos es una de las nociones fundamentales en que se basan los sistemas orientados a objetos, por lo que la redefinición de su semántica es una herramienta muy poderosa. Esta idea, no sólo

subyace bajo esta tesis, sino en otros sistemas como Merlin[AK95], ApertOS[Yok92] o MetaJava[GK97a], Sistemas Orientados a Objetos en los que, al igual que en este trabajo, cualquier cosa que pase se inicia gracias a la invocación de métodos.

2.1 Reflectividad en la invocación de métodos

Se habla de **reflectividad en la invocación de un método** *m* de un objeto del nivel base O_1 por parte de otro objeto del nivel base O_2, cuando la serie de acciones, potencialmente complejas, entre el objeto que envía y el que recibe el mensaje es **gestionada por metaobjetos** de O_1 y O_2.

2.1.1 Aspectos expuestos de la Invocación de Métodos

Los dos aspectos del paso y tratamiento de mensajes que normalmente componen el núcleo reflectivo de la invocación y ejecución de métodos son la **búsqueda y ejecución del método correspondiente al mensaje recibido**, acciones estas que se llevan a cabo del **lado servidor** en la recepción del mensaje en muchos sistemas implementados actualmente como CodA[McA95a, McA95b], MetaJava[GK97a], Merlin[AK95], SOM[IBM96a, IBM96b] o Smalltalk[FJ89, Riv96].

En esta tesis se propone también, para incrementar la flexiblidad del sistema, la **exposición del envío de mensajes en el lado del cliente**, como en el caso de Merlin[AK95].

La exposición del envío y recepción de mensajes, o lo que es lo mismo, la exposición del comportamiento de la máquina para la invocación de métodos, permite a los objetos del nivel base emisor y receptor, establecer protocolos privados de comunicación, mecanismos de búsqueda del objeto destino, definir un comportamiento específico ante errores o condiciones excepcionales, instalar temporizadores para establecer tiempos de respuesta o determinar si el paso de mensajes será síncrono o asíncrono[Hof99].

2.1.2 Meta-Objetos para Exponer la invocación de Métodos

En la propuesta que se hace en esta tesis, basada en el modelo de Meta-Objetos, cada objeto tendría asociado un meta-componente que denominaremos por ahora **Mensajero**.

El meta-componente mensajero introducido ya en el capítulo XIV, se compondrá entonces de dos meta-objetos, virtualmente independientes, cada uno de los cuales se encarga de uno de los aspectos de la gestión de mensajes, el envío o la recepción, y que dialogan mediante el paso de mensajes para llevar a cabo la invocación efectiva del método.

Al dividirse la funcionalidad en dos meta-objetos, cada uno de ellos podrá especializarse por separado dotando de mayor flexibilidad al entorno.

Estos dos meta-objetos recibirán el nombre de Mensajero-Emisor o **Emisor** y Mensajero-Receptor o **Receptor** para indicar, no sólo su funcionalidad sino también el origen del meta-objeto.

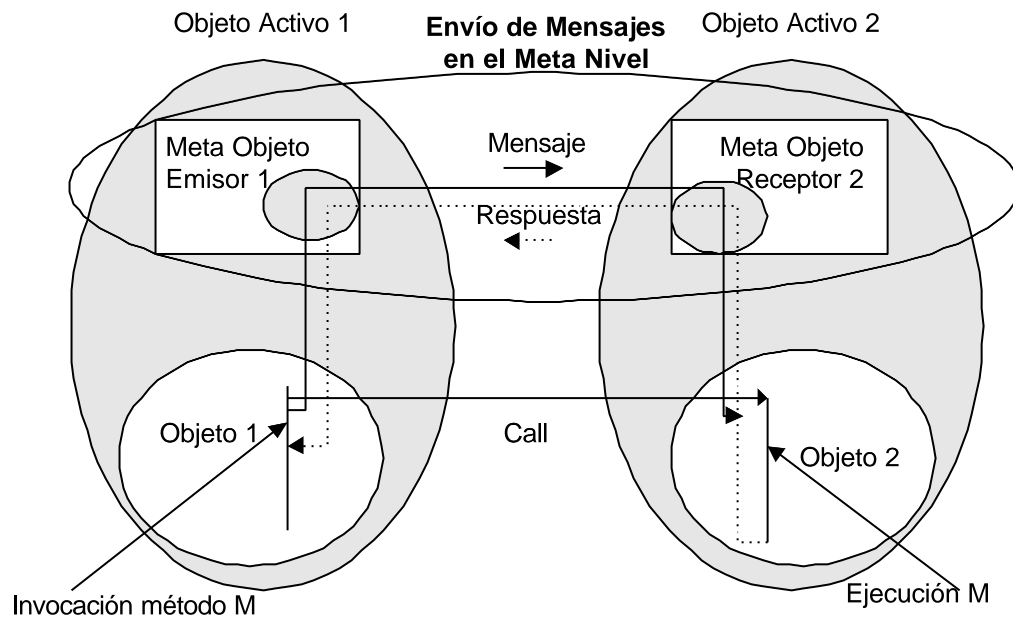


Figura 16.3 Composición Interna de un Objeto

2.1.3 Paso de Control del nivel Base al Meta-Nivel

La invocación de un método se inicia cuando, durante la ejecución de un método, un objeto ejecuta la instrucción **Call** de la máquina. Como consecuencia de esto, la máquina abstracta reacciona pasándole el control al meta-espacio asociado al objeto que envía el mensaje, como se vio en el Capítulo XV (ver figura 16.3).

Es fundamental que ambos sean equivalentes, es decir, las operaciones en el meta nivel para el envío de mensajes deben ser consistentes con la semántica de la operación Call o Send.

El paso al meta-espacio se realiza mediante la invocación de un método concreto en el meta-objeto Emisor, al que por simplicidad daremos el nombre **RCallE** (Reflejo de la parte Emisora de la instrucción que provoca su ejecución, Call).

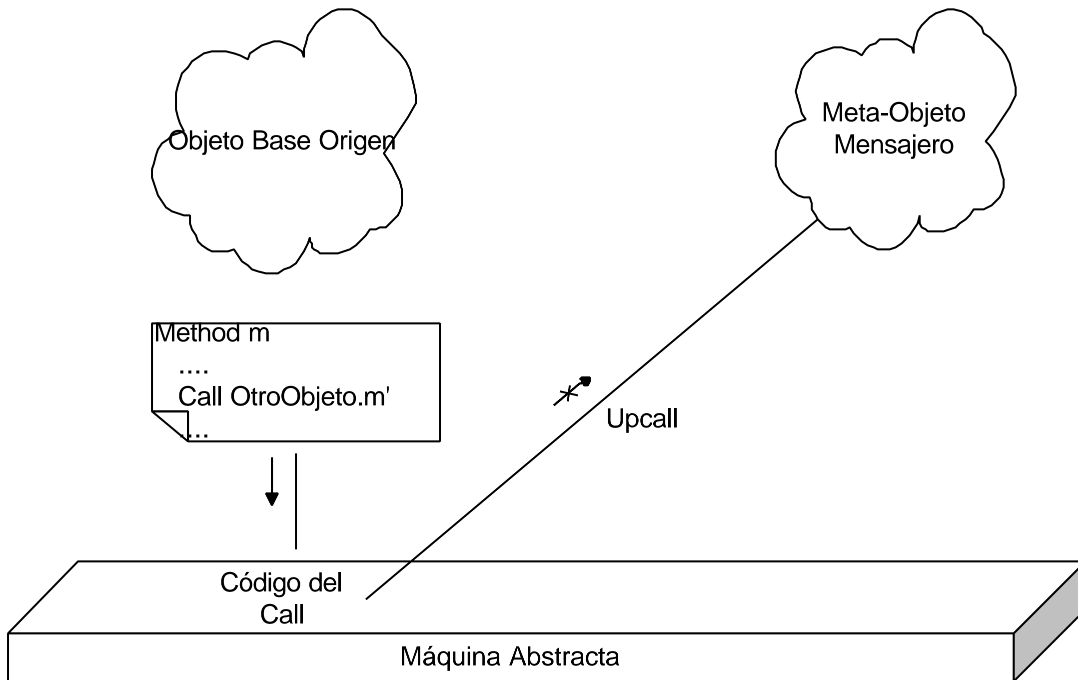


Figura 16.4 Instrucciones de Paso de Mensajes: Primitivas de Bajo Nivel y Traps al Sistema Operativo.

2.2 Meta-Objeto Mensajero-Emisor

El meta-objeto Emisor se encarga de realizar las acciones que el nivel base disponga para la invocación del método o envío del mensaje. Para ello, cuando un objeto del nivel base invoca un método de algún otro objeto del nivel base, o lo que es lo mismo, le envía un mensaje, el meta-objeto Emisor del objeto origen de la llamada, captura el envío y toma el control de la operación de envío o lo que es lo mismo, la máquina abstracta, en el proceso de interpretación de la instrucción invoca el método Call del meta-objeto Emisor.

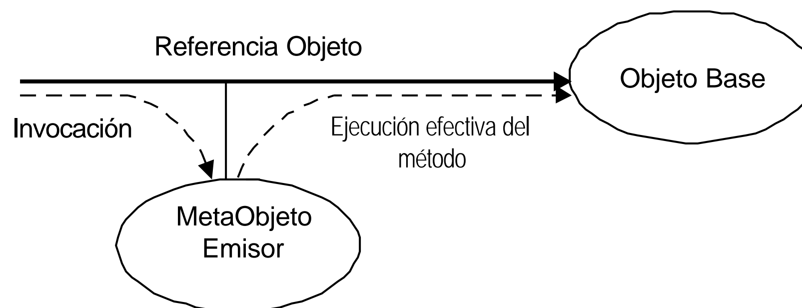


Figura 16.5 Invocación de Métodos. Trap al MetaObjeto Emisor

Este meta-objeto expone diversos aspectos del envío de mensajes que el objeto de nivel base puede adaptar a su entorno de ejecución, de tal forma que es posible personalizar las operaciones que este metaobjeto lleva a cabo tanto antes como después de realizar el envío.

En general, dado que los meta-objetos se codifican en el lenguaje de la máquina, se pueden implantar sus métodos de tal forma que su comportamiento quede totalmente adaptado a los requerimientos de las aplicaciones.

2.2.1 Objetivo del Meta-Objeto

El principal objetivo del sistema de invocación de métodos, máquina y meta-objeto, es conseguir que el método invocado se ejecute como se especifica en el objeto del nivel base.

Sin embargo, para dotar de flexibilidad al entorno, permite adaptar las acciones de invocación del método al entorno y objetos concretos de que se trate. Concretamente, el emisor podrá especializar su comportamiento para invocar un método en otro objeto, de forma independiente en varios aspectos de los que a continuación se citan algunos de ellos.

Para ello, el meta-objeto puede hacer varias operaciones antes y después de que la operación de envío se lleve a cabo de forma efectiva, es decir, antes y después de que se cree un nuevo hilo en el objeto destino para servir la invocación.

2.2.2 Mecanismos de Adaptación

Selección del Servidor

Cuando un objeto del nivel base invoca un método en otro objeto, el mensaje se envía, normalmente, al objeto indicado por el objeto base origen. Pero, en caso de problemas, que pueden ir desde la no disponibilidad del objeto servidor, hasta que éste rechace el mensaje, el meta-objeto puede decidir no transmitir el error al objeto del nivel base sino reintentar nuevamente la llamada más tarde o seleccionar otro servidor, en caso de que estén replicados, en función de algún conjunto de características indicadas por el objeto base (por ejemplo, se busca el receptor y, si no está en la máquina local, es posible dar un error, o buscarlo en la máquina destino).

La decisión se toma en función de la información disponible y de los criterios de decisión especificados por el nivel base en el momento de la configuración del meta-nivel, es decir, de la codificación particular que se haga de cada meta-objeto.

Gestión de Errores

En caso de problemas, como que el objeto destinatario no esté disponible, que este objeto rechace el mensaje o que, habiéndose establecido un tiempo máximo de espera para obtener respuesta este tiempo expire, la aplicación puede personalizar el meta-objeto emisor para que reaccione buscando una solución.

Acciones como seleccionar un objeto servidor alternativo o generar una respuesta al objeto base, son ejemplos de soluciones al problema. El meta-objeto considera la información existente acerca del contexto del objeto base y decide qué hacer con el mensaje.

Soporte para Características de Tiempo Real

El meta-objeto puede solicitar soporte de la máquina en forma de las tradicionales alarmas de UNIX para controlar el envío de mensajes. De esta forma, puede decidir asignar una alarma a cada mensaje para señalar puntos del tiempo en los que el meta-objeto debe intervenir.

Tales puntos pueden representar por ejemplo, time-outs de reconocimiento de llegada del mensaje o time-outs de espera de la respuesta. Si la señal llega antes que el evento esperado, el meta-objeto interviene para encontrar una solución. Si el evento que

se controla se produce antes de que la señal de alarma se dispare, el meta-objeto cancela la alarma.

Información para Estadísticas y Depurado

Una de las acciones más populares será la de registrar y mantener información estadística acerca del servicio de mensajes y registrar datos para posterior depurado, como pueden ser objeto destino de la invocación, tiempo de espera, y otros.

Información para Decisiones en el Meta-Nivel

El meta-objeto recolecta información acerca de la actividad del objeto que, posteriormente, se utiliza en los procesos de decisión. Saca provecho de la exposición del envío de mensajes para obtener datos como tiempo de respuesta, calidad de servicio, tiempo de comunicación, etc. que pueden ser útiles en aspectos como el balance de carga y otros.

2.2.3 El Meta-Objeto Emisor de cerca

A continuación se muestra una aproximación más detallada del comportamiento del emisor de mensajes. Estas acciones tendrían lugar cuando un objeto del nivel base invocase un método de otro objeto también del nivel base, es decir, cuando el objeto base ejecutase las instrucciones Call o Send.

Envío de mensajes: Ejecución de la Instrucción Call por parte de un objeto

1. Transición del nivel base al meta-nivel. El control se pasa al meta-objeto Emisor del objeto correspondiente mediante la invocación del método **RCallE**.
2. Se ejecutan las acciones del meta-nivel señaladas por el meta-objeto Emisor del lado origen del envío. Estas acciones son anteriores al envío efectivo del mensaje.
3. Avisar al objeto receptor mediante la invocación de su meta-objeto Receptor (**RCallR**). Esta acción equivaldría a poner el mensaje en la cola del Mensajero del receptor. Como las invocaciones a y entre meta objetos son síncronas, el meta objeto Emisor del origen se queda esperando la respuesta del meta objeto Receptor del destino.
4. Cuando terminen las acciones del lado destino, el meta objeto Emisor recupera el control. Si la llamada se produjo con una instrucción Call, esto sucede cuando termina la ejecución efectiva del método solicitado. Si se ejecutó la instrucción Send, el meta objeto Emisor recupera el control mucho antes. Concretamente, en cuanto se ejecute completamente el método RCallR del meta objeto Receptor.
5. En cualquier caso, una vez recupera el control, puede realizar acciones posteriores a la invocación efectiva del método.
6. Finalmente, devuelve el control al objeto base, es decir, se produce una transferencia de control del meta nivel al nivel base.

La única instrucción que se ejecutará en el objeto original será la instrucción Call/Send que, inmediatamente, efectuará una transición al meta-nivel y será el meta-objeto Emisor el que continúe.

En la implantación final el meta objeto Emisor, al igual que el Receptor y cualquier otro meta-objeto que sea necesario introducir, se escribirá en el lenguaje definido por la

máquina abstracta, lo que implica que se puede definir en cualquier lenguaje de alto nivel para el que se disponga de un compilador a tal lenguaje ensamblador.

De esta forma, este código podrá ser modificado por el programador según sus requerimientos de funcionalidad, eficiencia, etc.

2.2.4 Ejecución del Meta-Objeto Mensajero-Emisor

Para la ejecución de las tareas del meta-nivel, el objeto que realiza la llamada cederá parte de su tiempo al meta-espacio, tanto a su meta-objeto como al Mensajero del objeto que recibe la llamada. Así se procesan el envío y la llegada del mensaje inmediatamente.

Esta forma de actuar, simplifica enormemente el diseño del meta-objeto Mensajero, evitando que tenga que estar chequeando la cola continuamente para detectar nuevos mensajes, y en general del meta-espacio.

El retorno de resultados, al tratarse de objetos normales, es igual que en aquellos.

2.3 Meta-Objeto Mensajero-Receptor

Por su parte, el meta-objeto **Receptor** se encarga de realizar las acciones que el nivel base disponga cuando reciba una invocación a uno de sus métodos o llegada de un mensaje.

Para ello, cuando algún objeto del nivel base invoca un método de otro objeto del nivel base, o lo que es lo mismo, le envía un mensaje, el meta-objeto Emisor invoca el método correspondiente en el meta-objeto Receptor del objeto destino de la llamada, que toma el control de la invocación y realiza diversas acciones antes de pasar el control al nivel-base para la ejecución efectiva del método.

Este meta-objeto expone diversos aspectos de la recepción y ejecución de métodos que el objeto de nivel base puede adaptar a su entorno de ejecución, personalizando las operaciones que este meta-objeto lleva a cabo.

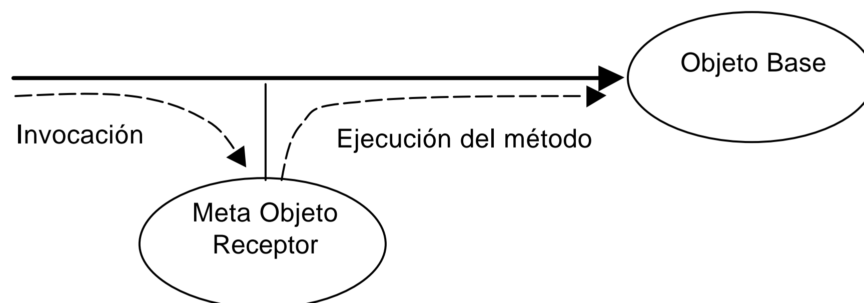


Figura 16.6 Invocación de Métodos. Trap al MetaObjeto Receptor

2.3.1 Ejecución del Meta-Objeto Receptor

La recepción de mensajes en el lado receptor se produce como consecuencia de la invocación del método **RCallR**(Reflejo de la parte Receptora de la instrucción Call) del meta-objeto Receptor por parte del meta-objeto Emisor.

La invocación de este método provoca que el meta-objeto Receptor reciba los mensajes y, a medida que estos mensajes llegan, el Receptor los acepta y analiza el contexto de ejecución del objeto para determinar la actuación a llevar a cabo.

Virtualmente, estos mensajes se colocan en una cola de mensajes (buzón) del meta-objeto Receptor asociado al objeto destino. El meta-objeto Receptor que recibe los mensajes, analiza la cola de mensajes para decidir qué hacer con los mensajes.

En la implantación final del entorno de ejecución, se prescinde del buzoneo de mensajes, por suponer una sobrecarga al obligar al meta-objeto Receptor a chequear continuamente la cola para detectar la llegada de nuevos mensajes.

Por el contrario, de la misma forma que el objeto origen cede parte de su tiempo a su meta-espacio para que envíe el mensaje, también se cede parte del tiempo al meta-espacio del objeto receptor para que reciba el mensaje.

Es decir, se cede tiempo para realizar la operación completa igual que si la ejecutase la máquina abstracta.

2.3.2 Objetivo del Meta-Objeto

El principal objetivo del meta-objeto Receptor es completar la funcionalidad del meta-objeto Emisor realizando las acciones de recepción de mensajes.

Juntos, Emisor y Receptor sobreescriben la instrucción Call de la máquina por lo que su semántica debe ser consistente con la de la instrucción Call, ya sea la invocación síncrona o asíncrona.

Además, con el fin de permitir la adaptación del entorno, el meta-objeto Receptor se codifica en el lenguaje de la máquina, al igual que el meta-objeto Emisor, lo que facilita la adaptación del mecanismo de invocación de métodos, consiguiendo así que el método invocado se ejecute como se especifica en el objeto del nivel base.

El usuario podrá especializar el comportamiento del meta-objeto receptor en varios aspectos, de los que a continuación se citan algunos de ellos.

Para ello, el meta-objeto puede hacer varias operaciones antes y después de que la operación de Recepción se lleve a cabo de forma efectiva.

2.3.3 Adaptación del Mecanismo de Invocación de Métodos: Análisis del Entorno de Ejecución

Objetivos del Análisis

Este análisis determinará si se acepta o rechaza el mensaje, o si se delega en algún otro objeto. Además, ante un fallo en la comunicación por ejemplo, la imposibilidad de servir el mensaje de forma inmediata, el objeto puede decidir devolver un error, esperar, o lo que considere oportuno. Incluso, puede mantener un diálogo previo con el emisor para buscar una solución de consenso[GK97a].

El hecho de que el mensaje sea aceptado, no implica que sea ejecutado inmediatamente como se verá en el meta-objeto Sincronizador.

Factores que influyen en el Análisis

Este meta-objeto considera varios factores del entorno de ejecución del objeto para el análisis del mensaje.

Entre ellos se encuentran el propio mensaje, el estado del objeto, el contexto de ejecución del objeto receptor, la disponibilidad de recursos (como uso de la máquina) y criterios de decisión y mecanismos internos del propio objeto así como información propia del meta-objeto para decidir qué hacer con el mensaje.

2.3.4 Selección del Método

Es uno de los mecanismos más importantes y utilizados para la adaptación del entorno de ejecución, en concreto, de la invocación de métodos.

Cada vez que el meta-objeto Receptor recibe un mensaje, debe tomar una decisión acerca de cómo conseguir que se ejecute el método correspondiente y, para ello, el objeto puede proporcionar más de un método que satisfaga el mensaje.

Cada método tiene características particulares como tiempo de ejecución, tipo de método y otras que el meta-objeto debe analizar para seleccionar el mejor método de acuerdo a las características particulares de la petición, la carga, etc.

2.3.5 Otras Acciones

Retraso de Ejecución

Ante una carga excesiva de trabajo o cualquier otra causa que impida el servicio inmediato de un mensaje por parte de un objeto, la actuación por defecto es retrasar el mensaje.

Posteriores análisis de la cola de mensajes ante mejoras en la carga, u otras razones, podrán decidir la ejecución de los mismos.

Pero el usuario puede decidir tomar otras medidas.

Delegación de Mensajes

Cuando un objeto no puede ejecutar el servicio demandado por diversas razones como el exceso de carga de trabajo, el meta-objeto Receptor puede decidir delegar la ejecución de este método en otro objeto aumentando sus posibilidades de ejecución.

2.3.6 El Meta-Objeto Receptor en profundidad

Una vez que el mensaje ha llegado al meta-objeto Receptor, este objeto llevará a cabo la siguiente secuencia de acciones, destinadas a cumplir la ejecución de la instrucción Call o Send.

Recepción de Mensajes:

1. Se invoca el método RCallR/RSendR del meta objeto Receptor. Normalmente, será el meta objeto Emisor el que lo haga.
2. El objeto Receptor estudia el mensaje recibido: origen, método solicitado, parámetros, etc.

3. Realiza las acciones determinadas por el usuario antes de decidir la ejecución efectiva del método.

   Analiza la carga de trabajo del objeto y, si es necesario, ejecuta el método TooLoad para determinar si el método se delega (DelegateMethod), se retrasa (DelayMethod) o continua su ejecución.

4. Si continua su ejecución, busca el método a ejecutar (LookupMethod).

5. Si la invocación es síncrona, se ejecuta la instrucción Call de la máquina que comienza la ejecución del mismo, sin invocar al planificador. Se produce así una transferencia de control del meta nivel al nivel base. El meta objeto se queda esperando en una invocación síncrona la finalización de la ejecución del método.

   Cuando termine, puede realizar más acciones antes de retornar el control al meta objeto Emisor del objeto origen.

6. Si la invocación es asíncrona, el meta objeto Receptor ejecuta la instrucción Send que devolverá el control enseguida al meta objeto y que invoca el método NewMethod del planificador para informar de la existencia de un nuevo hilo de control. A partir de ahora, el meta objeto Receptor devuelve el control al meta objeto Emisor del objeto origen.

2.4 Información del Meta-Nivel

El meta-nivel debe mantener la información necesaria para tomar las decisiones que el nivel base considere oportunas, dotando al sistema de cierta capacidad de adaptación ante variaciones en el entorno.

Esta información consta, tanto de información relacionada con el contexto de ejecución del objeto como información del meta-nivel en sí.

Información acerca del objeto

Esta información es acerca del objeto en sí, incluyendo información del nivel base y del meta-nivel, desde los criterios de decisión ante determinados eventos, hasta información estadística acerca de la ejecución de métodos, información para depurado, etc.

- **Método/Acción**: Contiene las acciones a realizar para cada mensaje que se recibe. Se accede cada vez que se recibe una invocación a un método para averiguar qué hacer. Como ejemplos de posibles acciones a realizar están la ejecución directa, la delegación, etc., considerando el contexto de ejecución y las características del mensaje.

- **Información de Ejecución de los métodos**: Contiene datos acerca del tiempo de ejecución de cada método como WCET (Tiempo de Ejecución en el peor Caso), BCET (Tiempo de Ejecución en el Mejor Caso) y OCET (Tiempo de ejecución Siendo Optimista) que se corresponde con un valor de tiempo en el que un gran porcentaje de las invocaciones terminarían. Estos datos son importantes cuando, en una aplicación, es necesario tomar decisiones que involucran aspectos temporales, como en aplicaciones con restricciones de tiempo real. Se definen durante el desarrollo pero se actualizan durante la ejecución.

Información acerca del Contexto

Esta información es recolectada por los meta-objetos durante la ejecución y utilizada posteriormente, a la hora de tomar decisiones. Entre la información de contexto, se puede citar:

- **Tiempo de respuesta**: tiempo de comunicación estimado con otros objetos. Este tiempo lo calcula el objeto que realice el papel de cliente en una invocación en función del momento de envío del mensaje y la recepción de la respuesta.
- **Quality of Service (QoS)**: Contiene medidas del QoS de otros objetos.

2.5 Ejecución de los métodos en el Meta-Nivel

Para llevar a cabo las acciones del meta-nivel asociadas a la invocación de métodos, el meta-objeto Emisor y Receptor deben recibir el control de la máquina. Para ello, cuando por alguna razón debe ejecutarse alguno de los dos, el nivel base invocará de forma síncrona el método adecuado del meta-nivel cediendo además, tiempo de ejecución (hand-off) al meta-espacio para procesar el envío y la llegada del mensaje inmediatamente.

Esta forma de actuar, simplifica enormemente el diseño del meta-objeto Mensajero (emisor y receptor), evitando que tenga que estar chequeando la cola continuamente para detectar nuevos mensajes, y en general del meta-espacio.

2.5.1 Diálogo en el Meta-Nivel

Los meta-objetos Mensajeros de origen(Emisor) y destino (Receptor) de la invocación dialogan mediante el paso de Mensajes para determinar la secuencia concreta de acciones a realizar en la invocación efectiva del método o paso de mensaje.

Para ello, ejecutan la instrucción Call con objeto destino el meta objeto correspondiente y el método solicitado (Call MObjeto.método(parámetros)).

Fin de la recursión

Dado que los meta-objetos Mensajeros son a su vez objetos, el paso de mensajes entre ambos debería también, estar gestionado por otros meta-objetos mensajeros.

Esta recursión termina con la implantación de la primitiva Call en la máquina que realiza la invocación efectiva del método solicitado, como se vio en el Capítulo XV.

2.6 Ventajas de la reflectividad en la invocación a métodos

En general, la exposición de la invocación de métodos es una de las herramientas más potentes para poder personalizar el entorno de ejecución de un objeto, con todas las ventajas derivadas de ello.

Sin embargo, hay algunos aspectos de particular importancia que se ven muy beneficiados por la posibilidad de codificar un entorno de paso de mensajes personalizado.

Soporte adaptado a distintos tipos de Lenguajes

En general, el soporte en tiempo de ejecución que necesitan los lenguajes varía de unos a otros en distintos aspectos.

Uno de tales aspectos es la gestión de errores. En caso de que se produzca un error, la opción por defecto es lanzar y propagar una excepción. Sin embargo, esto serviría de poco para aquellos lenguajes sin capacidad para manejar excepciones.

Es posible definir en el meta-nivel una variable global, tipo **errno**, y codificar los meta-objetos de forma que procesen las excepciones reflejándolas en la variable errno y propagando así los errores al nivel base.

Personalización de la Invocación para mayor eficiencia

Otro aspecto muy importante es el del retraso en la ejecución de los métodos. Tales retrasos se producen en el sistema por múltiples causas (invocación de un método en un objeto que todavía no está instanciado, en el caso de una invocación asíncrona, carga del objeto destino, etc.).

En todos estos casos, es posible codificar los meta-objetos para que, conociendo el entorno de ejecución de los objetos del nivel base puedan, no retrasar la ejecución de los métodos, sino tomar otro camino como devolver un aviso al cliente para que lo reintente más tarde, buscar un servidor alternativo, etc.

Restricciones de Tiempo

También es posible imponer en el meta nivel restricciones de tiempo a los métodos. Esto sería muy útil en entornos de tiempo real.

Facilita la creación de entornos distribuidos y persistentes al tratar todo ello en el meta-nivel

La creación de un entorno distribuido se ve beneficiada por la exposición de la invocación de métodos ya que es posible ocultar en los meta-objetos toda la maquinaria necesaria para invocar métodos remotos de forma transparente, localizar objetos, gestionar errores derivados de la ausencia de un objeto servidor, en cuyo caso, en lugar de devolver un error, se podría intentar buscar otro, o incluso crear uno nuevo y enviarle el mensaje, etc.

3 Modelo de Objetos Activo en el MetaNivel: Control de la Concurrencia Interna de los Objetos

Muchas aplicaciones se benefician del uso de objetos activos que sirven sus métodos concurrentemente para mejorar su QoS (del inglés, *Quality of Service*), en lugar de utilizar objetos pasivos, que ejecutan sus métodos en el hilo de control que realiza la llamada.

Sin embargo, dado que los objetos ejecutan sus métodos simultáneamente, es necesario sincronizar los accesos a sus métodos y datos cumpliendo varias premisas básicas.

Control del Rendimiento

Es necesario asegurar que los métodos invocados en un objeto de forma concurrente y que no pueden ser servidos inmediatamente debido a razones como la necesaria preservación de la consistencia del estado interno del objeto, no bloqueen el objeto. Es decir, el objeto debe poder seguir procesando otros mensajes mientras espera la resolución de algún otro.

Para ello, una vez se decide ejecutar un método ante la invocación del mismo por parte de un objeto, se crea un nuevo hilo o contexto para ello que no interfiere con la ejecución de otros métodos.

Simplicidad en la sincronización

Muchas aplicaciones son difíciles de programar si los desarrolladores deben utilizar mecanismos de bajo nivel como las cerraduras o los semáforos de forma explícita en el código de los métodos cliente y/o servidor.

En general, es posible establecer un primer nivel de control de la concurrencia en el borde del objeto, ante la llegada de una invocación a un método. Este **control de grano grueso,** si se compara con la utilización de primitivas como las antes mencionadas, es, sin embargo, válido en una gran parte de los casos que necesitan serializar, de forma transparente, la ejecución de algunos métodos sujetos a restricciones de sincronización, cuando múltiples clientes acceden a un objeto.

Transparencia

Las aplicaciones deberían diseñarse para aprovechar el paralelismo subyacente en una plataforma hardware o software de forma transparente.

Para cumplir estas premisas, se propone en esta tesis desasociar la invocación de los métodos de un objeto de su ejecución, de forma que, si la primera se gestionaba en el meta-nivel por medio de una serie de meta-objetos, la segunda también tiene su correspondencia en el meta-nivel con el meta-objeto Sincronizador y el meta-objeto Planificador.

3.1 Reflectividad en el Control de la Concurrencia

Se habla de **Reflectividad en el Control de la Concurrencia** cuando la sincronización de la ejecución de métodos se lleva a cabo, al menos en cierta medida, en el meta-nivel y no de forma explícita en el nivel base mediante el uso de objetos cerradura, semáforos, etc., en el código de los métodos.

En este apartado, correspondiente a la Sincronización Intra-Objetos, la atención estará centrada en describir la estructura y funcionalidad del meta-objeto Sincronizador.

3.1.1 Objetivos

Cualquier solución OO al problema de la sincronización debe cumplir las siguientes propiedades[Rit97].

Encapsulación.

Se requiere que la parte de sincronización en la ejecución de métodos de un objeto se encuentre colocada o asociada con el objeto, en lugar de estar dispersa como parte del código de sus clientes.

Esto coloca la responsabilidad de sincronización en el objeto, en lugar de depender de los clientes.

Extensibilidad

Se requiere también la abstracción de las políticas de sincronización. No es posible hallar una política óptima para todas las situaciones, sino que depende del dominio de ejecución del objeto. Las políticas deben ser personalizables.

Modularidad

Requiere la separación de la sincronización del objeto de la funcionalidad del mismo. Esta ortogonalidad permite cambiar la política de sincronización del objeto sin repercusión en otros componentes.

Reusabilidad

Requiere poder reutilizar por separado tanto el código secuencial como el código de sincronización. Debería ser factible reutilizar independientemente ambos.

3.1.2 Aspectos Expuestos

Emisor y Receptor de mensajes determinan la forma en que se lleva a cabo la interacción y la sincronización entre objetos.

Aún habiendo recibido un mensaje – acción esta de la que se encargan los dos meta-objetos anteriores –, un objeto puede, posteriormente, rechazar, ignorar o retrasar su ejecución.

Este aspecto de la ejecución de métodos, más relacionado con el modelo de objetos, que determina la forma en que los objetos activos se comportan ante la recepción de mensajes, se expone también a la manipulación del usuario. Es decir, se permite al usuario personalizar la forma en que los objetos **proceden desde que reciben el mensaje hasta que ejecutan el método adecuado**.

Este aspecto, que forma parte del modelo de objetos, está codificado internamente en el propio funcionamiento de la máquina abstracta y se expone con el fin de poder modificar, al menos en parte, ese modelo.

La exposición del comportamiento del objeto ante la aceptación – que no recepción – de un mensaje, permite a las aplicaciones establecer **distintos comportamientos para un mismo objeto base** dependiendo del entorno de ejecución en el que se vaya a utilizar, **sin tener que modificar el objeto base** en sí.

Así, se pueden definir objetos monitores, serializadores, etc., comportamiento este que se controla en el meta-nivel del objeto.

3.1.3 Meta-Objetos para Exponer el Control Interno

En la propuesta que se hace en esta tesis, basada en el modelo de Meta-Objetos, cada objeto tendría asociado un meta-objeto que denominaremos **Sincronizador**.

Este meta-objeto que, al igual que los anteriores, fue introducido en el capítulo XIV, se encargará de exponer los aspectos de la **Sincronización Intra-Objetos**. Cada aplicación podrá personalizar su propio entorno, o incluso personalizar el entorno de cada objeto particular, dotando a cada uno de ellos de un comportamiento personalizado.

La desasociación código base/código del meta-nivel, junto con la posibilidad de modificar el meta-objeto de un objeto, ofrecen la posibilidad de modificar el modelo de objetos subyacente, proporcionando una gran flexibilidad al entorno.

3.2 El Meta-Objeto Sincronizador

De la misma forma que el objeto base tenía asociado un meta-objeto Emisor y Receptor, cada objeto tiene también en su entorno un **meta-objeto Sincronizador** que determina la política de sincronización de grano grueso del objeto base, política esta que se lleva a cabo en el borde del objeto, ante la recepción de un mensaje o invocación de un método.

Así, si el meta-objeto Receptor definía el protocolo de recepción de mensajes, el Meta-Objeto Sincronizador define la **Política de Aceptación de Mensajes**, lo que significa que determina cómo y cuándo se ejecutan los distintos métodos de un objeto.

Para ello, cuando se solicita la ejecución de un método, una vez recibido, el meta-objeto Receptor invoca el método correspondiente en este meta-objeto – **ExecNewMethod(mensaje solicitado, origen del mensaje, otros parámetros)** – para determinar la posible Aceptación del mensaje.

El meta-objeto Sincronizador consulta la especificación del comportamiento del objeto definida por el usuario – o se guía por el comportamiento por defecto – estudiando las distintas restricciones de ejecución antes de determinar si la ejecución de ese método es adecuada o no.

Esto dependerá del entorno de ejecución actual del objeto: qué métodos ya están en ejecución (lo que no implica que tengan asociado en este preciso instante un procesador físico), qué métodos están a la espera de ejecutarse y qué métodos están parados en alguna condición de sincronización.

De esta forma, la máquina abstracta cede otro de sus procesos de ejecución al control del usuario, siendo ahora este el que determina el comportamiento del objeto. Así, dado que este meta-objeto expone parte del modelo de objeto del objeto de nivel base, es posible adaptar su entorno de ejecución, personalizándolo para una mayor eficiencia.

En general, dado que los meta-objetos se codifican en el lenguaje de la máquina, se pueden implantar sus métodos de tal forma que su comportamiento quede totalmente adaptado a los requerimientos de las aplicaciones.

3.2.1 Objetivos del Meta-Objeto

El objetivo principal del meta-objeto sincronizador es exponer aspectos de la sincronización del objeto de forma que se pueda especificar una política de sincronización de grano grueso, adaptable a cada objeto.

Por defecto, es la propia máquina abstracta la que se encarga de definir este aspecto del modelo de objetos, obligando a serializar la ejecución de los métodos de los objetos.

Sin embargo, para dotar de flexibilidad al entorno, permite relajar esta norma tan restrictiva adaptándola al entorno y objetos concretos de que se trate.

De esta forma, cuando un objeto recibe un mensaje (meta-objeto receptor), esto no significa que vaya a ser procesado inmediatamente ni que vaya a ser obligatoriamente retrasado, sino que el meta-objeto receptor pasará el control al meta-objeto sincronizador.

Este será el encargado de analizar el mensaje y, a la luz del estado de ejecución actual del objeto y, según la política de sincronización definida, procederá a su ejecución o no.

3.2.2 Análisis de Mensajes o Política de Servicio de Invocaciones

Sincronización de Grano Grueso: Política para aceptar los nuevos mensajes

Es posible definir qué restricciones afectan a la ejecución de los métodos de un objeto, bien de forma global a todos los métodos, bien de forma individual a cada método.

Para ello, el usuario define, bien para el objeto, bien para el método, las condiciones para su ejecución. Dado que el meta-objeto Sincronizador se puede escribir en un lenguaje de alto nivel que tenga intérprete al lenguaje de la máquina, el usuario es libre de definir esas condiciones utilizando para ellos los métodos y estructuras de control que desee.

El meta-objeto sincronizador ofrece en su interfaz el método **ExecNewMethod** que utilizará esos métodos para determinar una acción: Ejecución o Retraso del método invocado.

Aquellos mensajes que el meta-objeto Receptor ha decidido recibir son analizados por el meta-objeto sincronizador que, dependiendo de la política de sincronización, del método invocado y del contexto actual de ejecución del objeto, los encolará en la cola de invocaciones todavía no ejecutables – retrasa su ejecución – o determinará que debe ser ejecutado inmediatamente, lo que supone la creación de un nuevo hilo de ejecución o contexto.

En caso de que decida retrasar la ejecución del método solicitado, el emisor del mensaje puede continuar su ejecución o bloquearse a la espera del resultado, dependiendo del mecanismo de paso de mensajes elegido. Hasta este momento, cede su tiempo de ejecución al meta-nivel para que este procure las acciones necesarias para la ejecución efectiva del método.

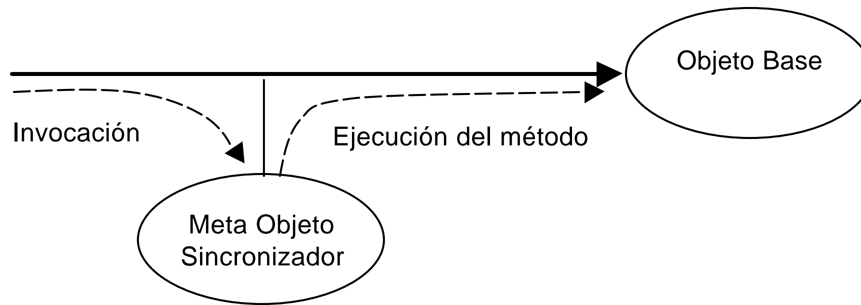


Figura 16.7 Reflectividad Implícita para la Sincronización de Grano Grueso

Prioridades a los métodos

Aunque la política de sincronización de métodos sea la principal misión del meta-objeto Sincronizador, en general, se dedica también a gestionar la ejecución de los mensajes.

Esto incluye asignar prioridades a los métodos. Estas prioridades pueden estar basadas en el nombre del método, el emisor o el receptor del mensaje.

3.3 Ejecución del Meta-Objeto Sincronizador

El meta-objeto Sincronizador no analiza la posible ejecución de métodos únicamente ante la recepción de un método sino que, periódicamente, ante la ocurrencia de eventos que puedan modificar el contexto de ejecución del objeto, vuelve a analizar la lista de mensajes a la espera, y elabora nuevamente una lista de posibles métodos a ejecutar.

Si hay disponible algún mensaje para uno de estos métodos, puede determinar su ejecución, en cuyo caso, sería necesario un nuevo análisis de la cola de mensajes, debido al cambio de estado del objeto por esta última decisión. La tarea termina cuando no es posible ejecutar ningún mensaje en la cola de espera y han sido todos examinados.

La relación de sincronización entre el objeto base y el meta-objeto Sincronizador, se lleva a cabo ejecutando determinados métodos de este último ante ciertos eventos que se produzcan en el contexto de ejecución del objeto del nivel base durante la ejecución normal de sus métodos.

Esta conexión existente entre un objeto del nivel base y su meta-objeto sincronizador supone que, ante distintas circunstancias acaecidas en el objeto, como pueden ser la invocación de un método(ya mencionada) o la finalización de otro, provoca la ejecución de métodos en el meta-objeto sincronizador.

Los métodos del meta-objeto sincronizador pueden ejecutarse, bien por medio del mecanismo de invocación de métodos, bien mediante el paso implícito de control al meta-nivel por parte de la máquina abstracta.

La ejecución de métodos en el meta-objeto sincronizador se produce siempre como resultado de uno de los siguientes eventos: **Invocación de un Método, Suspensión de un método, Reanudación de un método y Finalización de un método**.

3.3.1 Reflectividad Explícita

En el primero de los casos – invocación de método – estamos hablando de Reflectividad Explícita y suele producirse cuando otro meta-objeto, como el meta-objeto receptor invoca un método del meta-objeto sincronizador, como **ExecNewMethod**.

El caso más claro es cuando se invoca un método en un objeto base, en cuyo caso el meta-objeto receptor invoca el método ExecNewMethod.

En este caso, será el meta-objeto Receptor el que con más frecuencia invoque los métodos de la interfaz del Sincronizador con el fin de provocar el análisis del mensaje recibido, no ya según el punto de vista del mensaje, sino de la idoneidad de las condiciones actuales de ejecución del objeto para proceder a su ejecución.

Reflectividad Implícita

Otra posibilidad de ejecución de métodos del meta-objeto Sincronizador es cuando la máquina abstracta pasa el control a un método del meta-objeto ante la ocurrencia de algunos sucesos como la finalización, reanudación o suspensión de un método, que provocan un cambio en el contexto de ejecución del objeto y, por tanto, que el control pase al meta-nivel.

En el meta-nivel es posible definir, además del método ExecNewMethod anterior, otros métodos que gestionan la finalización, reanudación o suspensión de un método, como son **EndMethod**, **RestartMethod** y **StopMethod**.

Cada vez que se produce alguno de los eventos señalados, se invoca el método correspondiente del meta-objeto, indicando el método que produce el evento y el origen de la llamada, para evitar bloqueos mutuos.

En cualquiera de los dos casos, modificando los métodos o el mapeo de los mismos, cambiamos la política de sincronización del objeto.

De esta forma se simplifica considerablemente la tarea de derivar una subclase a partir de una clase concurrente ya que el código base no cambia y únicamente sería necesario modificar el código de sincronización.

3.3.2 Sincronización exhaustiva de Métodos

La política de sincronización puede involucrar a métodos individuales, para lo que, para cada método M del objeto base pueden existir métodos con nombre entryM, exitM, stopM, resumeM, IentryM, IexitM, IstopM, IresumeM que representan el inicio o finalización de la ejecución del método M, el hecho de que M se quede bloqueado, o que se reanude su ejecución, y lo mismo si esto sucede desde un método del mismo objeto. Estos métodos se invocarían desde los anteriores, si se definen.

Intentos similares se están llevando a cabo con el lenguaje Java[GK97b].

3.3.3 Ejecución del método

Sea como sea, el meta objeto sincronizador puede determinar la ejecución del método. En este caso, invoca el método correspondiente mediante la ejecución de la instrucción Call. La máquina abstracta se encarga de realizar las acciones correspondientes como son crear un nuevo hilo, crear el objeto hilo correspondiente e

invocar al objeto planificador, si la llamada fuese asíncrona, o iniciar la ejecución inmediata del método con el tiempo cedido por el objeto origen, si la llamada fuese síncrona.

3.4 Ventajas de la Reflectividad en el Control de la Concurrencia dentro de los objetos

Separación de asuntos

La separación de la funcionalidad base y la sincronización en un objeto hace más sencilla la programación de la aplicación base, al no tener que estar pendientes de la concurrencia en este mismo nivel.

Control más sencillo de la sincronización

El hecho de que la responsabilidad de la ejecución dependa o esté asociada directamente con el objeto base, hace más sencillo controlar el correcto acceso concurrente al mismo, al no depender del código de los clientes, cuya negligencia supondría un claro atentado contra la integridad del objeto.

Problema de la herencia

Se soluciona el problema de la herencia al definir por separado la funcionalidad base del objeto de sus restricciones de sincronización permitiendo especializar por separado cada una de ellas, evitando así la necesaria modificación del código fuente al derivar una clase, en el caso en que ambos estaban mezclados.

4 El sistema de Planificación del Sistema Operativo

La ejecución en el sistema integral aquí definido está encapsulada dentro de los objetos, y la lleva a cabo de forma implícita la máquina abstracta.

Sin embargo, dejar en manos de la máquina toda la gestión de la ejecución de métodos, fundamentalmente la planificación, supone codificar una política de planificación única e inamovible para cualquier aplicación que se ejecute.

La idea subyacente es que, para implementar una estrategia de planificación óptima, el planificador debe tener información acerca del comportamiento de la aplicación y debe ser capaz de interactuar con ella para determinar el estado de la misma y de los hilos que se están ejecutando[KR94].

Aunque en la mayoría de los sistemas operativos tradicionales e incluso modernos, es común que la estrategia de planificación esté integrada en el núcleo y el usuario no pueda cambiarla, sistemas operativos como Mach ya permiten cierta flexibilidad en la planificación, desde dar pistas acerca de la planificación de los hilos[Bla90], hasta poder cambiar la política de planificación

Por ello, se decide en esta tesis hacer accesible a las aplicaciones parte de las tareas de gestión de la ejecución, en concreto, la planificación.

De esta forma, la gestión de la ejecución se lleva a cabo parte en la máquina abstracta (creación, destrucción de hilos, ejecución de instrucciones, interrupciones de

tiempo), y parte en el Sistema Operativo, por medio de objetos del sistema (que forman parte del espacio de objetos), que serán responsables de la gestión de la ejecución de las aplicaciones y que serán personalizados por el usuario.

4.1 Planificación a Varios Niveles

Esta política de planificación puede verse desde dos niveles distintos.

Planificación Global

La existencia de varios procesadores y multitud de objetos que pretendan ejecutar sus métodos, da lugar a la necesidad de una política de planificación que asigne tareas a procesadores o procesadores a tareas.

Este planificador global puede estar codificado a bajo nivel dentro de la propia máquina abstracta, lo que daría lugar a una política de planificación fija e inamovible, que es lo normal en los sistemas operativos de propósito general usuales.

La reflectividad de la planificación, expone este comportamiento de la máquina y lo representa con un objeto en el meta-nivel. Este objeto puede instanciarse en tiempo de arranque de una de las muchas clases de planificadores que pueden existir en el sistema. Incluso puede especializarse una clase nueva, derivada de alguna otra.

La máquina abstracta pasará el control a este meta-objeto que tomará la decisión de qué tarea ejecutar de todas las que estén listas.

Planificación Local

Por otra parte, dentro de los propios objetos concurrentes, pueden existir varios métodos en ejecución. Puede ser necesario establecer una ordenación parcial dentro del propio objeto que dé prioridades a unos métodos sobre otros, que decida qué otro método ejecutar en caso de que el método actual en ejecución de este objeto ceda el control de la máquina, etc.

4.2 Reflectividad en la Planificación

Se habla de **reflectividad en la planificación** cuando la política de planificación de tareas está expuesta al usuario y puede ser modificada, incluso en tiempo de ejecución. Para ello, la política de planificación se codifica en un meta-objeto que formará parte del entorno en tiempo de ejecución de los objetos, al igual que ya hacen otros Sistemas Operativos como Apertos[Yok92].

4.2.1 Aspectos Expuestos de la Reflectividad en la Planificación

La mayoría de las aplicaciones actuales, tanto las aplicaciones multihilo tradicionales, como las aplicaciones basadas en objetos, necesitan ser capaces de definir una estrategia particular para sus hilos (servidores de ficheros, servidores de red, ...) por varias razones: cumplir los requisitos no funcionales impuestos a la aplicación, mejorar su eficiencia, etc., aspectos estos que deben verse reflejados, de alguna forma en la política de planificación.

De esta forma, un objeto puede recibir e incluso aceptar mensajes y, finalmente, puede determinar retrasar su ejecución o incluso rechazarla.

De entre los aspectos expuestos en la planificación, destaca, fundamentalmente la Política de planificación aunque otros como el Balance dinámico de carga también se derivan de ello.

Política de Planificación: Necesidades específicas de objetos o grupos de objetos

En ocasiones, un objeto –fundamentalmente aquellos objetos pesados que actúan como servidores en el sentido tradicional del término–, o grupo de objetos – normalmente aquellos cuya ejecución está relacionada y que colaboran para lograr un fin común – determinan que la política de planificación por defecto es totalmente inadecuada para sus necesidades, como ocurre con muchas aplicaciones en los entornos tradicionales.

Por ejemplo, en un entorno donde la planificación se realiza siguiendo una estrategia FIFO, un objeto servidor puede decidir servir sus peticiones siguiendo algún tipo de prioridad.

Para poder atender este tipo de situaciones, es necesario que los objetos de usuario sean capaces de imponer de alguna forma sus necesidades de planificación, indicando la política adecuada para la ejecución de sus métodos.

Necesidades de entorno: Adaptación Dinámica

En otras circunstancias, es posible que la política de planificación sea válida en el momento de su definición pero que, a lo largo de la vida del sistema, deje de serlo, lo que, claramente, provocará una degradación del rendimiento. Aspectos como el Balance de carga deben controlarse dinámicamente y pueden provocar modificaciones una vez comenzada la ejecución de una aplicación.

Así, si el área de instancias es escasa y se están produciendo continuas faltas de objeto, es posible que el planificador decida no planificar parte de sus objetos para no causar faltas de objeto, o puede utilizar quantums de tiempo más largos, o incluso puede determinar la migración de un objeto a otra máquina si considera que el coste de la migración puede afrontarse – Balance de Carga Global del Sistema.

Es más, una aplicación por su cuenta, puede suspender temporalmente a algunos de los hilos de sus objetos si se percata de falta de recursos, o puede determinar una estrategia de planificación diferente a las demás – Balance de Carga Local a una aplicación.

Estos cambios, que se producen en función de la evolución del entorno de ejecución de cada objeto y del sistema en sí, requieren la posibilidad de modificar la estrategia de planificación en tiempo de ejecución, con el fin de adecuarse a las nuevas circunstancias del entorno.

4.2.2 Meta-Objetos para la Planificación

Igual que en los casos anteriores – el comportamiento del objeto ante la ejecución concurrente e invocación de métodos – la planificación es uno de los aspectos de la máquina abstracta que esta tesis propone exponer al exterior, elevándolo al nivel de objeto con el fin de permitir su adaptación.

Para ello, se propone definir un Meta-Objeto nuevo en el entorno del objeto que denominaremos **Meta-Objeto Planificador**, que realizará las tareas propias de su nombre.

Todo objeto del nivel base tiene asociado un planificador, bien sea el planificador por defecto o uno definido *ex professo*. Sin embargo, la relación no tiene por qué ser necesariamente uno-a-uno.

Se trata más bien de que un meta-objeto planificador forme parte del entorno de varios objetos del nivel base, similar a lo que propone Apertos[Yok92].

El Meta-Objeto Planificador se creará a petición del usuario como instancia de una clase de entre las que forman la jerarquía de planificadores. Si en un momento determinado las circunstancias del entorno cambiasen, sería posible cambiar a su vez este planificador por otro más adecuado.

4.3 Jerarquía de Planificadores

Existen dos jerarquías de planificadores en el sistema operativo propuesto: la **jerarquía de planificadores en tiempo de programación** que, gracias al uso de la herencia permite definir un amplio espectro de políticas de planificación reutilizando gran parte del código existente, y la **jerarquía de planificadores en tiempo de ejecución**, que determina la cesión de control entre objetos.

4.3.1 Jerarquía de Clases de Planificación: Reutilización del Código

El sistema operativo propuesto incluye una jerarquía de clases para el sistema de planificación[BS88, RF77]. La herramienta de diseño más importante a la hora de diseñar la jerarquía de planificadores es la **herencia de clases**. De esta forma, se desarrolla una jerarquía de clases que encapsulan las distintas políticas de planificación existentes. Así, se permite implementar las decisiones de diseño como un conjunto de subclases especializables según las políticas de ordenación de los procesos y la utilización o no de requisamiento.

La utilización de la herencia como herramienta básica de desarrollo para crear familias de sistemas operativos es una noción ampliamente extendida. Un ejemplo muy claro es el sistema operativo Choices [CJM+89].

Esta jerarquía tendrá como raíz una clase abstracta que representa el Sistema de Planificación Universal (**USS**) que define la interfaz estándar para las instancias de sus subclases concretas.

Cuando se arranca el sistema operativo, es posible elegir uno o varios planificadores de cualquiera de las clases concretas que existen en la jerarquía. La elección se hará en base a la política de requisamiento y ordenación deseadas.

Adición de Nuevos Planificadores: Ampliación de la Jerarquía

De esta forma, se ofrece además al usuario la posibilidad de especializar un planificador a su medida y **aumentar la jerarquía ya existente**.

A medida que la jerarquía crece en profundidad y anchura, **la relación superclase-subclase debe mantener la relación existente entre los algoritmos de planificación**.

Así, por ejemplo, si a la jerarquía anterior se añade una nueva clase SJF (*Shortest-job-First*), debería ser como superclase de la clase SRT (*Shortest-Remaining-Time*) ya

que esta última es una combinación de las prioridades del algoritmo SJF con el concepto de requisamiento.

Anatomía de un objeto planificador

Fundamentalmente, un objeto planificador consta de una lista de referencias a planificar – los elementos sobre los que se aplica la política definida – y una serie de métodos comunes a todo planificador cuya implantación determina la política del mismo.

La lista de elementos sobre los que se aplica la política de planificación son, claramente, los hilos listos para ejecutar en el objeto u objetos de cuyo entorno forme parte el meta objeto planificador.

Dado que los hilos son objetos internos, esta lista estará formada por objetos hilo que exponen los hilos internos, tal como se vio en el Capítulo XV.

Por su parte, los métodos fundamentales que forman la interfaz del meta objeto planificador son **ScheduleNext**, **enqueue** – ambos con el significado habitual – y el método **IsEmpty**, que determina si tiene algo que planificar o no.

Además, cada usuario puede definir planificadores todo lo sofisticados que desee o necesite, añadiendo nuevos métodos a estos, para proporcionar funcionalidad auxiliar.

Jerarquía Inicial

Inicialmente, el sistema operativo proporciona una pequeña jerarquía de planificadores como la que sigue.

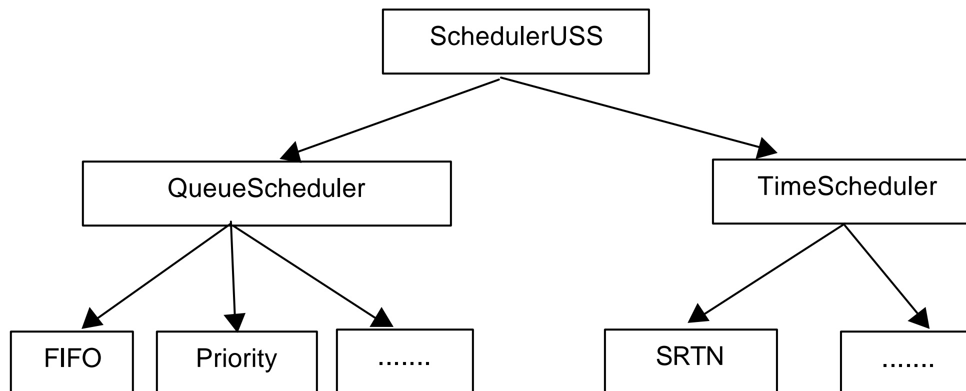


Figura 16.8 Pequeña Jerarquía de Planificadores ampliable mediante herencia

4.3.2 Jerarquía de Meta-Objetos Planificadores en Tiempo de Ejecución

Tradicionalmente, los mecanismos de planificación de los sistemas operativos han sido bastante rígidos. Los sistemas operativos tradicionales controlan el reparto de los recursos (entre ellos el tiempo de CPU o de ejecución) utilizando un esquema de planificación fijo.

Frecuentemente, por no decir siempre, han soportado únicamente una política de planificación y además, esa política era fija e inamovible. Como mucho, eran capaces de soportar unas cuantas clases de planificación distintas cuya implantación formaba parte

del núcleo del sistema operativo. Algunas veces se proporcionan variantes de esta política básica, como varias clases de planificación a las que se asignan hilos con diferentes propósitos.

Sin embargo, incluso estas variantes suelen estar integradas en la implantación del sistema y no pueden ser adaptadas fácilmente a las necesidades de las aplicaciones individuales[FS96].

Sin embargo, la planificación en los sistemas operativos modernos juega un papel fundamental en la eficiencia, la adecuación y la justicia de un sistema, especialmente para aplicaciones distribuidas multi-hilo y aquellas construidas siguiendo un esquema cliente/servidor [RK96].

Existen pues varias razones para crear una jerarquía de planificadores[KR94].

Planificación a Nivel Usuario con Poca Sobrecarga

En primer lugar, un planificador para un conjunto de objetos relacionados que formen una aplicación, **facilita el establecimiento de una política de planificación a nivel usuario**, independiente del planificador del sistema, sin la sobrecarga que suponía la estrategia de hilos usuario/hilos núcleo del Sistema Operativo.

Flexibilidad

En segundo lugar, puede resultar útil tener una estrategia de planificación separada para los hilos de un grupo de objetos dentro de una misma aplicación, con lo que se hace necesario un segundo planificador por debajo del planificador de la aplicación mencionado antes.

La posibilidad de tener un planificador general, junto con la posibilidad de añadir otros planificadores para aplicaciones, dota de una gran flexibilidad al sistema.

A continuación se verá este apartado con más profundidad.

4.4 Planificación: Jerarquía de Planificadores en tiempo de ejecución

El Sistema Operativo propuesto en esta tesis, ofrece una estructura de planificación basada en un número indeterminado de niveles de planificación, aunque, es posible simplificar la idea a **dos niveles** lo que facilitará la exposición y comprensión del concepto de **Jerarquía de Planificadores** sin pérdida de generalidad.

Los dos niveles virtuales en que puede dividirse la planificación en el sistema operativo propuesto son la **planificación de objetos** y la **planificación de meta-objetos**, idea adoptada también por Apertos[Yok92] y su antecesor Muse [YTT89].

4.4.1 Planificación de Objetos

Planificador: Meta-Objeto del Meta-Espacio

Cada metaespacio o entorno de objetos contiene un planificador para aquellos objetos a los que da soporte, de tal forma que los objetos del nivel base son planificados por el metaobjeto con el que estén asociados. Este metaobjeto puede implementar una política de planificación genérica, sin utilizar características propias y particulares del

entorno para tomar sus decisiones, o bien, por el contrario, puede utilizar información proporcionada por la aplicación relativa al entorno de ejecución al que está asociado, como la relación existente entre distintos métodos, etc.

Independencia de la Política de Planificación de distintos meta-objetos

Es necesario resaltar que la planificación de objetos en un meta-objeto es independiente de la planificación en otros meta-objetos. Esto permite proporcionar a cada objeto o conjunto de objetos el planificador más adecuado, asociando los objetos por aplicaciones, grupos de objetos con un interés común dentro de una aplicación, objetos individuales o incluso objetos que formen parte de distintas aplicaciones.

Aunque la planificación a este nivel puede definirse en **tiempo de programación** o **compilación**, aplicando mecanismos como la herencia de otras clases, los usuarios también pueden definir su propia estrategia de planificación, adecuada para ejecutar su aplicación.

Es más, los usuarios pueden cambiar dinámicamente las estrategias de planificación de objetos cambiando los objetos planificadores, en **tiempo de ejecución**.

Así, un objeto o grupo de objetos puede decidir controlar su planificación creando un planificador especializado y enlazándolo en la jerarquía de planificadores. Ideas similares han sido estudiadas en otros sistemas operativos como VINO[SS95]

Por tanto, el Sistema Operativo propuesto, ofrecerá a los objetos de usuario la posibilidad de asociar un planificador con el fin de especificar una política concreta de planificación para sus métodos, aunque la relación no tiene que ser necesariamente uno-a-uno.

4.4.2 Planificación de Meta-Objetos

Jerarquía de Objetos Planificadores

Los objetos del nivel base pueden asociarse y determinar la utilización de una política de planificación específica, común a todos ellos, implementada por un meta-objeto que implemente la funcionalidad de planificación. Este meta-objeto será una instancia de alguna de las clases de planificadores definidas en la jerarquía descrita en un apartado anterior.

Los **meta-objetos planificadores** son a su vez objetos. Por tanto, la ejecución de los meta-objetos planificadores en una misma máquina estará planificada por un meta-objeto planificador y así sucesivamente.

Son, por tanto, necesarios meta-objetos planificadores que planifiquen a otros meta-objetos planificadores más cercanos a los objetos del nivel base.

Se crea así una **jerarquía de planificadores** que se pasan el control unos a otros. De esta forma, especializando el comportamiento del planificador, este se ajustará a los requerimientos específicos de una aplicación. Así, se proporciona el mecanismo necesario para dotar al sistema de flexibilidad mediante la sustitución de servidores de planificación, de forma similar a MACH o Chorus.

La Raíz de la Jerarquía

Esta jerarquía de objetos planificadores se termina con un planificador básico, implementado en la máquina abstracta, que implementará una planificación tipo RR con requisamiento con el fin de evitar el monopolio de la máquina.

La máquina abstracta gestiona además una serie de excepciones, del inglés *traps*, como son las interrupciones temporales o la creación de nuevos hilos con prioridad mayor al hilo en ejecución.

4.5 El Meta-Objeto Planificador

Dentro de la jerarquía de planificadores definida, pueden distinguirse dos niveles en cuanto a la implantación interna: los planificadores de más bajo nivel, más cercanos a la ejecución, y los planificadores de los meta-objetos.

4.5.1 Planificadores de bajo nivel

Internamente, cada uno de los meta-objetos planificadores de bajo nivel tendrá asociada una lista de hilos de ejecución, de uno o varios objetos, que pelean por el control de la máquina para ejecutar las instrucciones del método al que están ligados.

Cada vez que este meta-objeto planificador reciba el control de la máquina, mediante la **invocación del método ScheduleNext**, ejecutará las acciones determinadas por la política de planificación que implementa, para elegir, entre todos los hilos, el más adecuado para la ejecución.

Gracias a la exposición de la clase thread, activar un nuevo hilo es tan sencillo como invocar el método, **start** de esta clase, que provocará el inicio de la ejecución o reanudación del método representado en el thread.

La invocación del método **start** se realiza de forma síncrona por parte del meta-objeto planificador, que, evidentemente, no podrá planificar otro hilo hasta que el actual libere o le requisen la máquina.

Estos meta-objetos también se ven invocados por otras causas. La más común de ellas es cuando el **meta-objeto sincronizador** decida que se dan las condiciones necesarias para que la ejecución de un método invocado se lleve a efecto. En este caso, el meta-objeto sincronizador **invoca el método Enqueue** de este meta-objeto planificador, con el fin de que esta petición tenga oportunidades de ser ejecutada en algún momento.

4.5.2 Resto de Planificadores

La planificación de los meta-objetos planificadores en la misma máquina viene planificada a su vez por otros meta-objetos denominados meta-meta-objetos planificadores. Aunque la jerarquía podría extenderse lo que fuese necesario, los meta-meta-objetos tendrían un comportamiento e implantación muy similar, excepción hecha del **planificador raíz de la jerarquía**.

Internamente, cada meta-meta-objeto está compuesto de una cola de meta-objetos planificadores y un conjunto de métodos – **ScheduleNext**, **Enqueue** – que implantan una política de planificación eligiendo el siguiente meta-objeto a planificar y añadiendo nuevos planificadores a la jerarquía.

Cada meta-planificador recibe el control de manos de otro meta-planificador o directamente, de la raíz de la jerarquía, mediante la invocación de su método ScheduleNext. Esto provoca la elección de un nuevo meta-objeto que planificar al que, posteriormente se le cederá el control mediante la invocación de su método ScheduleNext.

Una vez cedido el control, el meta-planificador, al igual que los demás meta-objetos planificadores, no puede planificar ningún otro meta-objeto aunque, eso sí, puede ejecutar otros métodos como encolar nuevas peticiones.

Tanto los meta-planificadores como los meta-objetos planificador pueden implantar la política de planificación que deseen, cuyo ámbito se restringirá a los hilos o planificadores a los que dé soporte.

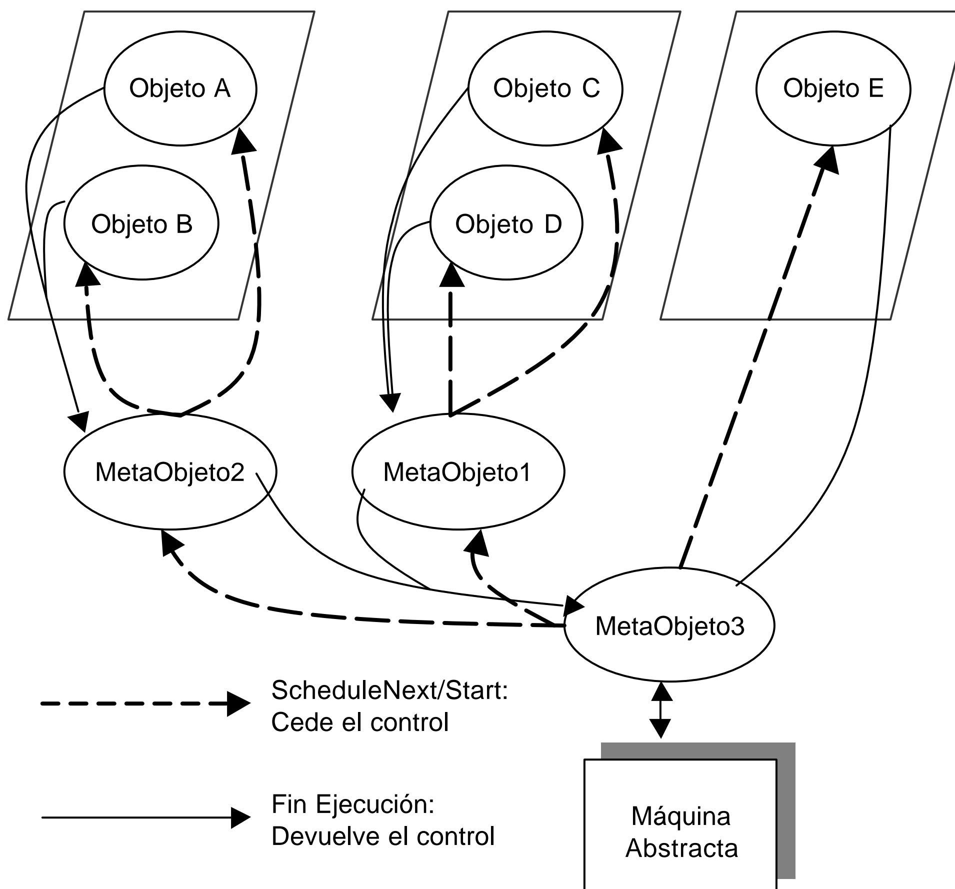


Figura 16.10 Planificación mediante varios Planificadores

4.5.3 La Raíz de la Jerarquía

La raíz de la jerarquía, el planificador global implementado en la máquina abstracta, termina la recursividad.

Básicamente, este meta-objeto reparte quantums de tiempo a los meta-objetos planificadores de primer nivel.

4.6 Transferencia de control entre los planificadores

La transferencia de control entre planificadores se hará siempre siguiendo la jerarquía formada, comenzando por el planificador raíz, implementado en la propia máquina abstracta, y siguiendo hasta las hojas de la jerarquía, que serán los planificadores más cercanos a los objetos, aquellos que planifican los hilos de ejecución.

4.6.1 Invocación de Métodos y Retorno

La transferencia de control de un planificador a uno de sus hijos en la jerarquía y el retorno al padre en la jerarquía, se realiza utilizando el mecanismo de paso de mensajes síncronos y retorno de los mismos.

Cuando un planificador reciba el control, lo hará por medio de la invocación de su método ScheduleNext.

Este método realiza las acciones necesarias para codificar una política concreta de planificación eligiendo, de entre todas las opciones que tenga, el hilo o planificador más adecuado.

Activación de un hilo

En el primer caso, invocará el método start síncronamente, lo que, al ser una invocación de métodos entre meta-objetos – el planificador y el meta-objeto que refleja la ejecución, ya visto en el capítulo anterior – será procesada utilizando la primitiva **Call** implementada en la máquina, que comenzará a ejecutar el hilo correspondiente.

Como la invocación se realiza de forma síncrona, el meta-objeto planificador no podrá continuar planificando, aunque sí se permite que reciba más peticiones para encolar otros hilos de control.

El planificador puede recuperar el control por varias razones: finalización del hilo de control que hubiese planificado – nótese que si se cedió el control directamente en el paso de mensajes, saltándose el planificador en la invocación, también se saltará el retorno al mismo, retornando el control directamente al hilo emisor –, suspensión del mismo por cualquier causa o excepción de la máquina como finalización del tiempo correspondiente o aparición de un nuevo hilo más prioritario.

Activación de un Planificador

En el segundo caso, se invoca el método Schedule_Next del planificador elegido para activar.

Finalización o Suspensión

En cualquiera de los dos casos, la máquina se comporta igual que si el paso de mensajes se hubiera realizado entre dos objetos cualesquiera. Cuando uno de ellos termina su ejecución, retorna el control al objeto que realizó la llamada, en este caso, el planificador, que reanudará su ejecución planificando más hilos.

4.6.2 Excepciones

Además de la finalización y la suspensión del hilo actual, existen otras causas que pueden determinar el paso de control entre planificadores: las interrupciones de tiempo y la aparición de un hilo con más prioridad.

Interrupción de Tiempo

En caso de que, durante la ejecución de un método cualquiera se produzca una interrupción de tiempo, la máquina inicia la gestión de dicha interrupción.

Tras las acciones iniciales, destinadas a guardar el contexto en ejecución, pasa el control al planificador correspondiente, como si el hilo hubiese cedido voluntariamente el control.

El planificador puede continuar su ejecución, si su tiempo total no ha terminado, o bien finalizar su ejecución, retornando el control a su padre, como en cualquier otro retorno de una llamada a método síncrona.

Con ello, se permite un reparto de los recursos más justo impidiendo que un objeto monopolice la máquina.

Prioridades

Si la planificación se está realizando por prioridades, y el planificador recibe un mensaje Enqueue con parámetro un hilo más prioritario que el hilo actual, suspenderá la ejecución del hilo menos prioritario, método **suspend**, y comenzará la ejecución del hilo más prioritario.

5 Visión Dinámica del Sistema

A continuación se detalla la colaboración de los distintos meta-objetos para conseguir el comportamiento de objeto activo.

La ejecución de métodos en un objeto, que comienza con una invocación a método, es examinada por varios meta-objetos para decidir cuándo y cómo se ejecuta.

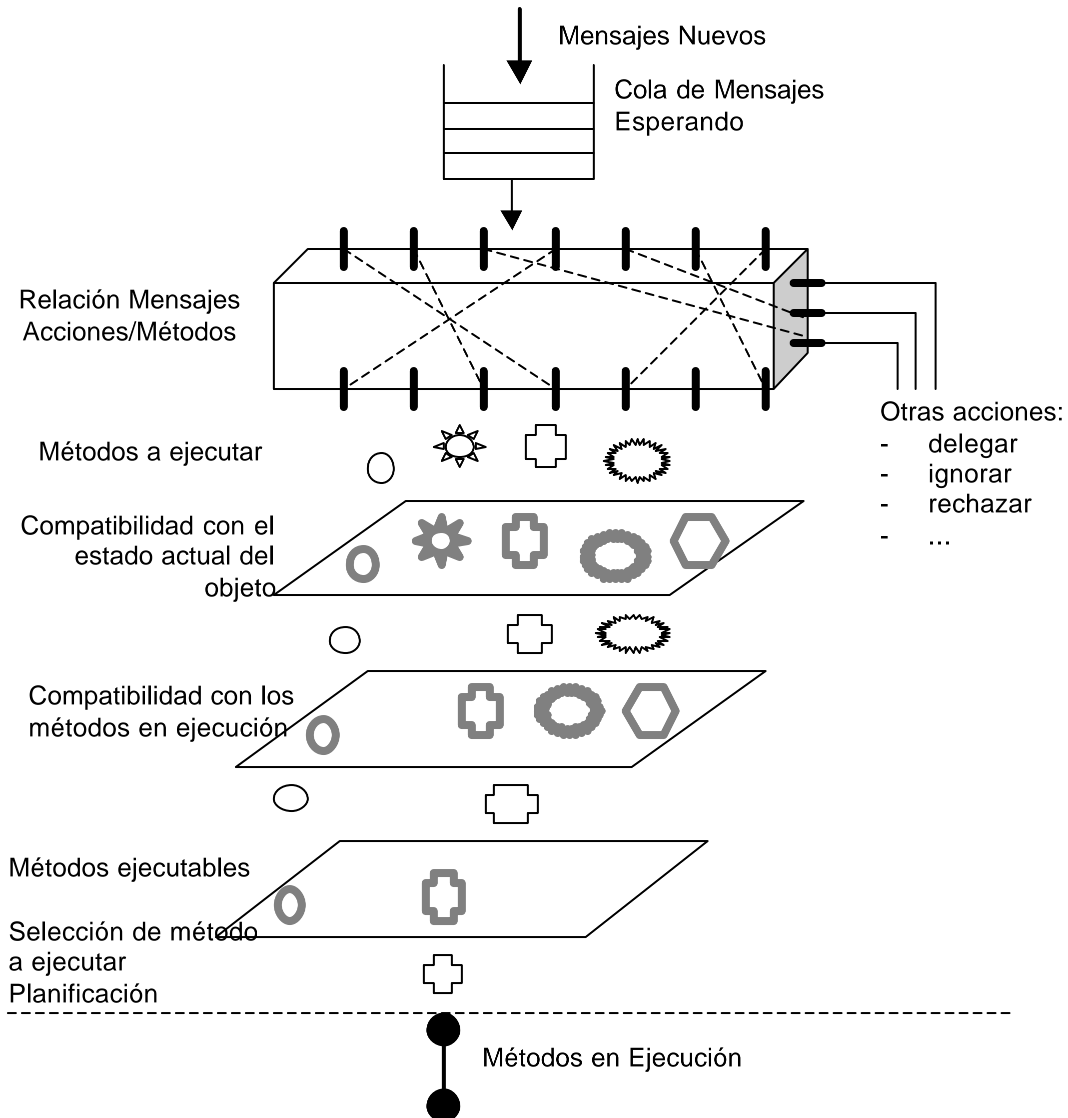


Figura 16.11 Llegada, procesamiento y ejecución de mensajes.

Para concluir este capítulo, se presenta una visión global del entorno activo de un objeto base, presentando sus meta-objetos (se presenta el caso más extremo de que un objeto tenga una instancia de cada meta-objeto) y las interacciones entre ellos en el caso más común de ejecución de un método.

- **Meta-Objeto Planificador**. Planifica la ejecución de los distintos hilos o contextos activos decidiendo cuál de ellos es el más adecuado en cada momento.
- **Meta-Objeto Emisor**. Se ocupa de la emisión de mensajes.
- **Meta-Objeto Receptor**. Se ocupa de la recepción de mensajes y de la propagación de excepciones como caso especial del retorno de una invocación.
- **Meta-Objeto Sincronizador**. Analiza los mensajes recibidos y decide cuándo se dan las condiciones adecuadas para crear un contexto o hilo para servir ese mensaje y pasárselo al planificador. O, por el contrario, cuándo es conveniente retrasar la ejecución de un método.

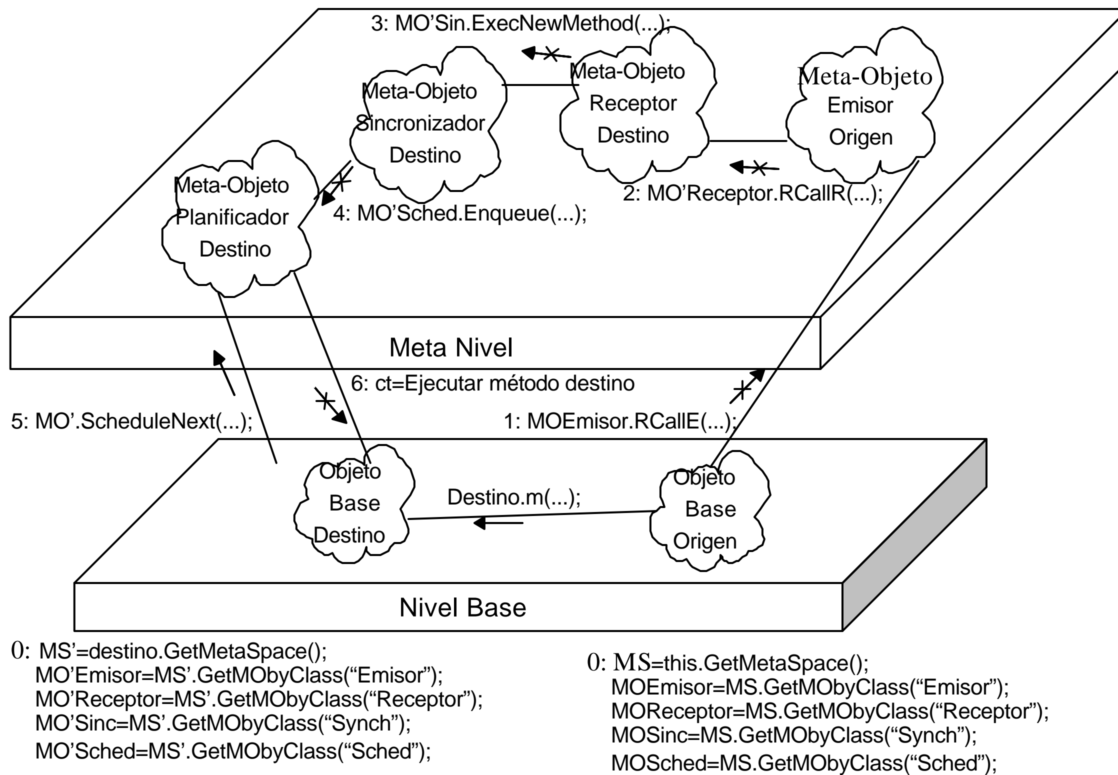


Figura 16.13 Llegada, procesamiento y ejecución de mensajes. Visión Dinámica

Aunque conceptualmente, cada objeto del nivel base puede solicitar a su meta-espacio la realización de una tarea, invocando cualquiera de sus métodos, la forma más usual en que el control de la ejecución pasa del nivel-base al meta-nivel es mediante la invocación de un método.

Así, la ejecución de la instrucción **Call** por parte de un método del nivel base, supone la transferencia de la computación en el nivel base de objetos a computación en su meta-nivel.

Cuando el meta-nivel termine su actividad con el fin de ejecutar convenientemente el método indicado en el objeto destino, el control retornará al nivel base.

Cada método en el meta-objeto se ejecuta de forma no interrumpible, lo que simplifica enormemente la transferencia de control. Esto se asemeja bastante a la gestión de llamadas al sistema por parte de Sistemas Operativos como UNIX.

0: Conseguir referencias a los metaobjetos es sencillo gracias a la reflectividad estructural básica de que está dotada la máquina, que permite la introspección y, con ello, interrogar a cada objeto acerca de su entorno.

1: Transferencia de control por parte del objeto base origen de la invocación a su meta-nivel. La invocación de un método en el nivel base, supone el paso de la computación del nivel base al meta-nivel. La meta-computación se inicia en el meta-objeto Emisor del objeto origen de la invocación que invoca el método correspondiente en el meta-objeto receptor del objeto base destino.

2: El meta-objeto Receptor dialoga con el Emisor con el fin de recibir el mensaje.

3: El meta-objeto Receptor envía un mensaje al meta-objeto Sincronizador para que estudie la conveniencia de ejecutar o retrasar la ejecución del método.

4: Si la invocación fue síncrona y la ejecución del método solicitado es adecuada en este instante, se invoca el método correspondiente. Posteriormente, se retornará el control al objeto emisor como retorno de la cadena de llamadas.

4: En otro caso, el meta-objeto sincronizador lo notificará al meta-objeto planificador que encolará el nuevo trabajo en la lista de planificación. El meta-objeto planificador ejecuta el método Enqueue, retorna el control al sincronizador, y así siguiendo la cadena de llamadas.

5: En algún momento, la máquina abstracta cederá el control al planificador y este ejecutará su método ScheduleNext

6: Si elige el método invocado, mediante la referencia del sistema ct, el ciclo de ejecución de la máquina continuará ejecutando el método correspondiente del nivel base.

Capítulo XVII

Prototipo de Sistema Integral (Máquina Abstracta y Sistema Operativo) Orientado a Objetos

El prototipo de la máquina desarrollado, a pesar de no ajustarse del todo, dado su carácter de prototipo, a la arquitectura propuesta en esta tesis, supone una plataforma de estudio y experimentación muy valiosa, sobre la que desarrollar posteriores proyectos que acerquen cada vez más la arquitectura de la máquina a los principios perseguidos. También supone la introducción del Sistema Operativo y dota al conjunto de un mecanismo de interacción.

Son varias las ideas utilizadas en el diseño y desarrollo de esta máquina que se revelaron como fundamentales y, por tanto, se mantendrán en posteriores prototipos, mejorando, por supuesto, la posible implantación de las mismas.

1 Presentación del Sistema Integral: Máquina Abstracta Carbayonia y Sistema Operativo

Se describe a continuación, la filosofía de Diseño e implantación del prototipo de Sistema Integral OO exponiendo cuáles fueron los principios que más influyeron en la etapa de diseño y desarrollo y las ventajas derivadas de la aplicación de los mismos.

En esta aproximación, se intentó únicamente crear una plataforma de desarrollo sobre la que experimentar las ideas expuestas en esta tesis, tratando de comprobar la facilidad de integración de los conceptos aquí desarrollados, como el modelo de objetos activo o la reflectividad, con la construcción de un SIOO.

Se trata pues de una aproximación a la arquitectura buscada en la que se introducirán ideas que se mantendrán a lo largo de todas las versiones, como la aplicación de los principios de Orientación a Objetos o la arquitectura reflectiva para la flexibilidad.

Se apreciará que no se hizo hincapié en cuestiones de eficiencia, sino que se apostó claramente por un diseño interno sencillo, que pueda ser implementado rápidamente y de fácil comprensión para posteriores refinamientos que tratarán de adecuar la arquitectura al diseño final perseguido.

1.1 Aplicación intensiva de los Principios de la OO

Indudablemente, la aplicación del principio de OO al propio diseño interno o a la implantación final de la máquina y el Sistema Operativo es uno de los grandes aciertos de las distintas versiones de la máquina[Izq96]. Del diseño e implantación como objetos de la máquina y su comportamiento se derivan una serie de ventajas que hacen que este sea uno de los principios de diseño básicos a mantener en las posteriores versiones que se desarrollen.

1.1.1 Aplicación de los Principios de la OO en el Diseño del Sistema Integral

En primer lugar, se aplicaron los principios de la OO en el diseño de la máquina y del Sistema Operativo, de lo que se derivaron un conjunto de ventajas inherentes al diseño de software OO y otras derivadas del hecho de que ese software vaya a ser utilizado como base de un sistema integral compuesto por máquina abstracta y sistema operativo.

Si ya las primeras son sumamente ventajosas, como se deriva de otros trabajos como [Rus91a] que profundizan más en este tema, las segundas hacen que se considere las aplicación de la OO al diseño como uno de los grandes aciertos.

Mantenimiento de la Uniformidad

El diseño de la máquina se hace con el objetivo en mente de dar soporte a una única abstracción, que es la abstracción de objeto, y, por tanto, ofrecer únicamente, mecanismos de interacción entre objetos.

El hecho de que no se haga ninguna diferencia entre el soporte a las aplicaciones (objetos de aplicación) y el soporte al sistema operativo (objetos del sistema operativo) o incluso que la propia plataforma en sí se organice en torno a objetos (las áreas son objetos), fomenta la uniformidad en torno a un único concepto, el objeto.

De esto, se derivan ventajas como la **economía de conceptos** y, por tanto, la eficiencia a la hora de desarrollar aplicaciones o familiarizarse con la plataforma.

Abstracciones Robustas en Ambientes Concurrentes

El principio de encapsulación determina que el acceso al estado interno de los objetos se hace, únicamente, a través de los métodos que el mismo ofrezca. De esta forma, se simplifica la construcción de objetos de forma que aseguren su consistencia en ambientes, tanto secuenciales como concurrentes.

En este sentido, el prototipo implanta un modelo de objetos activo que determina que los objetos se comunican a través de mensajes y pueden implementar una política de concurrencia adecuada al entorno de ejecución.

1.1.2 Aplicación de los Principios de OO en la implantación del Sistema Integral

En segundo lugar, se aplicaron las tecnologías OO en la construcción de la máquina, su implantación utilizando lenguajes OO supone ventajas como la reutilización del código muy conocidas.

Portabilidad

Organizar el Sistema Integral OO en torno a un núcleo pequeño que ofrece la funcionalidad básica, ya sea este un micronúcleo o una máquina abstracta, y un sistema operativo, es un punto de partida adecuado para conseguir la portabilidad, como ya se manifestó en los micronúcleos[Gie90], en Java[Ven98], etc.

A su vez, la aplicación de los principios de OO en la construcción de un sistema, sea la anteriormente citada plataforma básica o cualquier otra, favorece la encapsulación de los detalles dependientes de la plataforma hardware subyacente en objetos que proporcionan una interfaz a la entidad que abstraen lo que permite diseñar el resto del sistema de forma que descanse sobre tal interfaz y no sobre los detalles físicos particulares.

Idealmente, estos objetos se reimplementarán cuando el resto del sistema migre a una plataforma física distinta sin necesidad de ninguna modificación al resto del sistema, lo que permite que las porciones del sistema independientes de la arquitectura física subyacente, como el sistema operativo que se construya sobre la máquina abstracta o las aplicaciones de usuario que se ejecuten sobre ella, sean reutilizados sin grandes problemas.

Los principios y técnicas de OO proporcionan un marco de desarrollo adecuado para lograrlo. Se pueden utilizar clases abstractas para definir la interfaz de tales objetos y clases concretas para definir la implementación concreta que se adapte a la arquitectura.

El polimorfismo permite al código usar cualquier clase concreta que implemente la interfaz como un objeto distinto.

Reutilización

De forma similar a la portabilidad, la utilización de la herencia facilita la reutilización del código y la posible construcción de variantes de la misma máquina o sistema operativo con pequeñas modificaciones del código.

En el caso de la máquina abstracta, el caso más claro es, posiblemente, la adición de nuevas instrucciones al conjunto de instrucciones de la máquina. Una clase abstracta **TInstruction** encapsula el funcionamiento de las instrucciones, con una operación exec que provoca la ejecución de la instrucción. De ésta derivan todas las instrucciones de comportamiento que redefinen el método exec para que realice las operaciones necesarias de acuerdo con la definición de la instrucción.

Como parte de la ejecución de un método, existirá una invocación de un método sobre un objeto. El simulador accederá a la instancia y a la clase a través de la referencia. La clase indicará el objeto método que representa al método invocado. Una vez localizado el método se llamará a la operación invokeMethod del mismo. Este proceso es siempre el mismo independientemente de que el método sea primitivo o de usuario. El resto del simulador sólo utiliza objetos abstractos TMethod, con lo que el uso de métodos primitivos es totalmente transparente.

En este momento es donde entra en acción el polimorfismo. Si el método era primitivo, el tipo verdadero del objeto método devuelto será el de un método primitivo, por ejemplo TMIntegerAdd, e invokeMethod llamará a la implementación de este método en TMIntegerAdd, que realizará de manera directa la suma de enteros utilizando la representación interna primitiva de los mismos en la máquina.

En caso de ser un método de usuario, se llamaría polimórficamente al invokeMethod de un TMUserMethod, que desencadenaría la ejecución normal de un método mediante la simulación del funcionamiento de las instrucciones de la máquina que lo componen.

En el caso del Sistema Operativo, se puede seguir el mismo principio aplicado, por ejemplo, a la jerarquía de planificadores que por defecto se suministra.

Separación de Política y Mecanismo

Es una característica imprescindible a la hora de construir sistemas extensibles[SG98]. De la misma forma que herencia y polimorfismo soportan la portabilidad permitiendo definir fácilmente interfaces que se implementan de múltiples formas, también dan soporte a la separación de política y mecanismo ofreciendo la posibilidad de representar las interfaces a las distintas políticas con clases abstractas que definen la interfaz e instanciarlas a distintos objetos.

Soporte para interfaces adaptables

Una de las formas de extender un sistema es añadir nuevos componentes a la interfaz que este sistema ofrece a las aplicaciones.

La interfaz ofrecida a las aplicaciones por un SIOO se define por el conjunto de objetos que, normalmente, un servidor de nombres, pone a disposición de la aplicación. Este conjunto puede cambiar dinámicamente a medida que las necesidades de las aplicaciones cambian, lo que se logra modificando el conjunto de objetos accesibles a las aplicaciones

1.1.3 Reflectividad

Representación de los elementos de la máquina mediante objetos

La aplicación intensiva de los principios de la OO en el diseño de la máquina y el sistema operativo dio como resultado una plataforma en la que todos los elementos que la componen se traducen en objetos y la interacción entre ellos es mediante el mecanismo de paso de mensajes. La representación de los elementos mediante objetos es el primer paso para dotar de reflectividad al sistema [AK95].

La reflectividad supone la existencia de objetos que realizan parte de la funcionalidad de la máquina, al mismo nivel que los objetos de cualquier aplicación.

Objetos de la máquina escritos en C++

La reflectividad supone que los objetos que representan aspectos de la máquina no deben diferenciarse de los objetos de usuario, lo que implica que deben estar escritos en el lenguaje de la máquina. A partir de esto se logra uno de los grandes beneficios de la reflectividad, construir un sistema flexible ya que, al ser la máquina objetos normales, esto permitirá modificarlos en cualquier momento y, con ello, modificar el propio comportamiento de la máquina.

En este prototipo algunos de estos objetos son objetos del programa, o sea, objetos C++, por tanto, no son modificables, aunque otros sí se introducen como objetos de usuario, lo que refuerza la tesis de que es factible un diseño completamente reflectivo del Sistema Integral.

Primer paso hacia una arquitectura reflectiva completa

A pesar de que algunos de los objetos de la arquitectura son objetos del programa y no objetos en toda regla, el diseño actual de la máquina como conjunto de objetos favorece en cierto modo la consecución de una máquina reflectiva ya que quedan identificados los elementos (objetos) que migrarán a nivel de usuario en posteriores versiones que completarán la arquitectura reflectiva.

El proceso finalizaría cuando estos objetos que ahora forman parte de la máquina (objetos C++) estuviesen escritos en Carbayón y la invocación de sus métodos por parte de la máquina se realizase por medio de una invocación a métodos y no como una llamada a procedimiento en una aplicación C++, al igual que sucede con otros objetos que ya se introducen como objetos de usuario.

Reflectividad Estructural

En este prototipo se implementa una reflectividad estructural dinámica que no sólo permite a un objeto acceder a los objetos del programa que representan su estructura en tiempo de ejecución para interrogar acerca de su representación interna, como sería el caso del método `getClass()`que permite interrogar a la máquina acerca de la clase de un objeto, o el método `getName()` que permite conocer el nombre de un objeto en tiempo de ejecución. También permite al objeto modificar su representación, sus métodos, sus referencias agregadas y asociadas, etc., en tiempo de ejecución.

1.2 Simplicidad

En el diseño del sistema integral, otro de los principales objetivos fue lograr una plataforma simple, por lo que, en las decisiones tomadas, prevaleció la simplicidad frente a otras consideraciones.

1.2.1 Aplicación de los Principios de OO

La aplicación de los Principios de OO a lo largo de todo el diseño facilita la consecución de ese objetivo de simplicidad, al aunar todas las abstracciones en una única, el objeto.

1.2.2 Mecanismo de Comunicación

Las decisiones tomadas en el mecanismo de comunicación también persiguen el ansiado objetivo de simplicidad aunque sopesando la influencia que puede tener en otros como la eficiencia.

Comunicación Síncrona

Siguiendo la política de simplicidad, a la hora de elegir un mecanismo de comunicación, se optó, en primer lugar, por un paso de mensajes síncrono.

Este modelo de invocación de métodos es, con mucho, el más sencillo posible además de tener otras ventajas, que ya se discutieron en el Capítulo VIII.

Comunicación asíncrona

Existen razones de peso que fuerzan la introducción de un modelo de comunicación asíncrono. La necesidad de introducir concurrencia en el modelo, sin caer en los

problemas de añadir nuevas abstracciones o determinar en la definición de la clase el nivel de concurrencia permitido para cada método, abre el camino a la introducción de una nueva instrucción Send, que ejecuta el método de forma asíncrona con el emisor.

Rendimiento en ambientes concurrentes

Se ha comprobado que, en ambientes concurrentes, la introducción de esta instrucción y su utilización suponen una considerable mejora en el rendimiento de las aplicaciones.

➔ **Comprobación Alex.**

➔ **las últimas pruebas realizadas por Alejandro, dieron mejor rendimiento que el paso de mensajes síncrono**

2 Presentación de la máquina abstracta Carbayonia

En este apartado se describe el prototipo de la máquina abstracta Carbayonia (simulador por software de la máquina) desarrollado como proyecto fin de carrera de la Escuela Superior de Ingenieros Informáticos de la Universidad de Oviedo. Se puede consultar los detalles de la implantación en [Rod00].

2.1 Diseño de la máquina como conjunto de objetos

La aplicación intensiva de los principios de diseño expuestos con anterioridad, da como resultado un prototipo en el que, todos y cada uno de los elementos que constituyen la máquina, desde los elementos arquitectónicos, como pueden ser las áreas de clases o instancias, hasta las instrucciones que definen el ensamblador de la máquina, se representan en el simulador mediante objetos(TClassArea, TInstanceArea, TInstruction, ..).

Cada uno de estos objetos definen una parte del comportamiento de la máquina mediante un conjunto de métodos.

2.1.1 Diagrama de clases general

El resultado de aplicar el principio de OO en la propia construcción de la máquina, es una estructura en tiempo de ejecución de la misma en la que se representa, por medio de objetos en tiempo de ejecución, la estructura del programa en Carbayonia. Esto se muestra en el siguiente diagrama general de clases del prototipo:

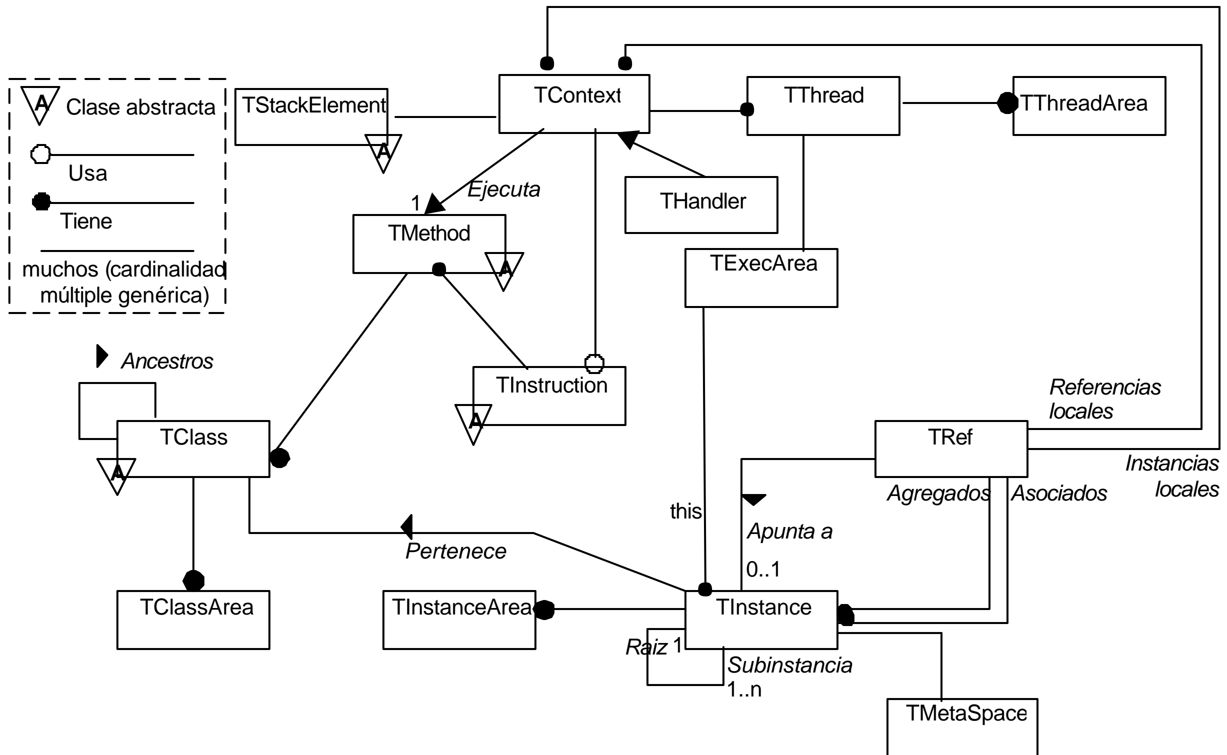


Figura 17.1 Diagrama de clases general del prototipo de la máquina Carbayonia.

2.2 Implantación de los elementos de la máquina: Uniformidad en la OO

Todos los elementos que componen el entorno se representan mediante uno o varios objetos, instancia de las clases que los describen. Así pues, a la hora de describir la implantación del SIOO es necesario describir la **jerarquía de clases del prototipo** y las relaciones entre ellas.

En el diseño e implantación de la máquina abstracta Carbayonia se aplicaron a fondo todos los principios de la OO siendo la herencia y el polimorfismo claves en el diseño. Esto facilita considerablemente la definición de la jerarquía de clases ya que, por un lado, la herencia permite definir un objeto abstracto, raíz de la jerarquía, a partir del cual, mediante refinamientos sucesivos, se pueden derivar distintas especializaciones. Por su parte, el polimorfismo, permite que el hecho de qué clase se trate sea totalmente transparente al resto del sistema.

2.2.1 Clases primitivas y clases de usuario

La máquina proporciona una implantación primitiva para ciertos elementos constituyentes de la máquina, que no podrán definirse en función de otras clases.

Estos elementos serán fundamentalmente las clases. Así, a las clases y métodos que son implementados directamente por la máquina se les denomina **clases y métodos primitivos**. Al resto, que se definen de manera normal a partir de otras clases se llaman **clases y métodos de usuario**.

El polimorfismo hace que este hecho sea totalmente transparente.

	Primitivas	de Usuario
Clases	No necesitan ser definidas en términos de otras clases. La máquina les da soporte directo.	Se definen en términos de otras clases (agregados y asociaciones), que les dan soporte.
Métodos	Son implementados directamente por la máquina. No necesitan cuerpo.	Son implementados por instrucciones normales de la máquina. Necesitan el cuerpo con estas instrucciones

2.3 Descripción de la Jerarquía de Clases Primitivas

En el caso del prototipo se implementarán de manera primitiva todas las **clases básicas** del modelo único: Object, MetaObject, Bool, Integer, Float, String y Array, junto con todos sus métodos. También el resto de las clases de apoyo: Semaphore, Stream, ConStream y FileStream.

El objeto abstracto raíz de la jerarquía, TClass, representa cualquier tipo de clase, y de él heredarán otros que representen clases primitivas (un TCxxx por cada clase primitiva) y de usuario.

A continuación se muestra la jerarquía relativa a las clases primitivas.

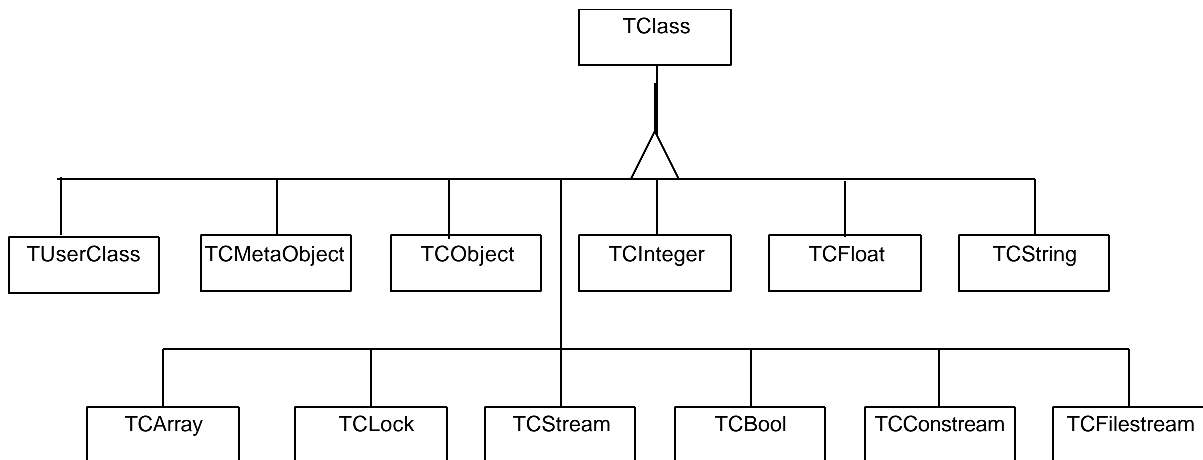


Figura 17.2 Jerarquía de TClass para las clases con implementación primitiva.

2.4 Descripción de los Objetos Instancias

De igual forma, se necesita una jerarquía similar para las instancias, para lo que se define una clase abstracta que representa una instancia en general (**TInstance**). Cada TCxxx para las clases tiene que tener su TIxxx en las instancias, tanto para las instancias de clases primitivas como para las de usuario. Así TCInteger trabajará con su TIInteger, TCUserClass con TIUserInstance, etc.

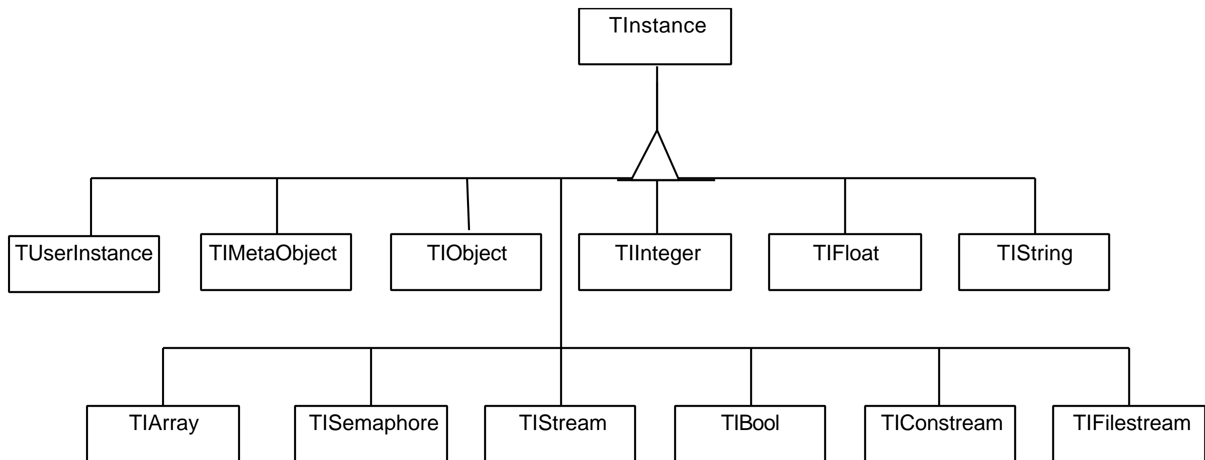


Figura 17.3 Jerarquía de TInstance para las instancias con implementación primitiva.

2.5 Descripción de los Métodos

El mismo procedimiento se aplica a los métodos. De la clase abstracta TMethod se derivarán los métodos de las clases primitivas y (la clase) para un método de usuario. En este caso existirá una clase para cada clase primitiva y de usuario. De cada clase primitiva derivan los métodos de la misma: de TMInteger derivan TMIntegerAdd, TMIntegerSub, etc.

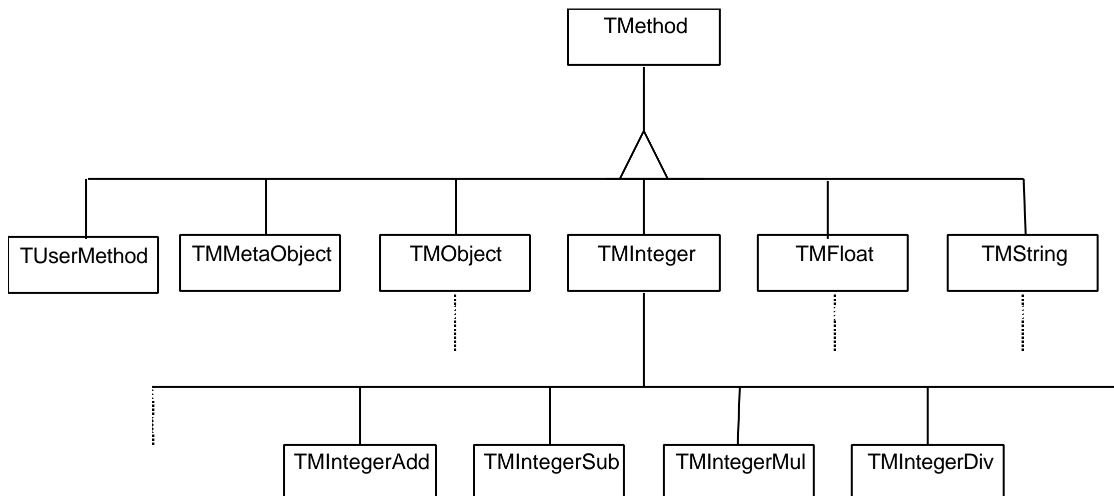


Figura 17.4 Jerarquía de TMethod para los métodos de las instancias con implantación primitiva.

2.6 Descripción de las Instrucciones

Las instrucciones de comportamiento para el cuerpo de los métodos también utilizan el mismo mecanismo, aunque aquí se podría considerar que todas estas instrucciones son primitivas, puesto que son las instrucciones de la máquina. Una clase abstracta **TInstruction** encapsula para el resto del sistema el funcionamiento de las instrucciones, con una operación exec que provoca la ejecución de la instrucción. De ésta derivan todas las instrucciones de comportamiento TInstrNew, TInstrHandler, etc. Cada una

redefinirá el método exec para que realice con las estructuras de representación de la ejecución (hilos y contextos) las operaciones necesarias de acuerdo con la definición de la instrucción. Por ejemplo, el exec de la instrucción TInstrJmp actualiza el valor del contador de método[1] (MC, *Method Counter*) para que apunte a la instrucción indicada.

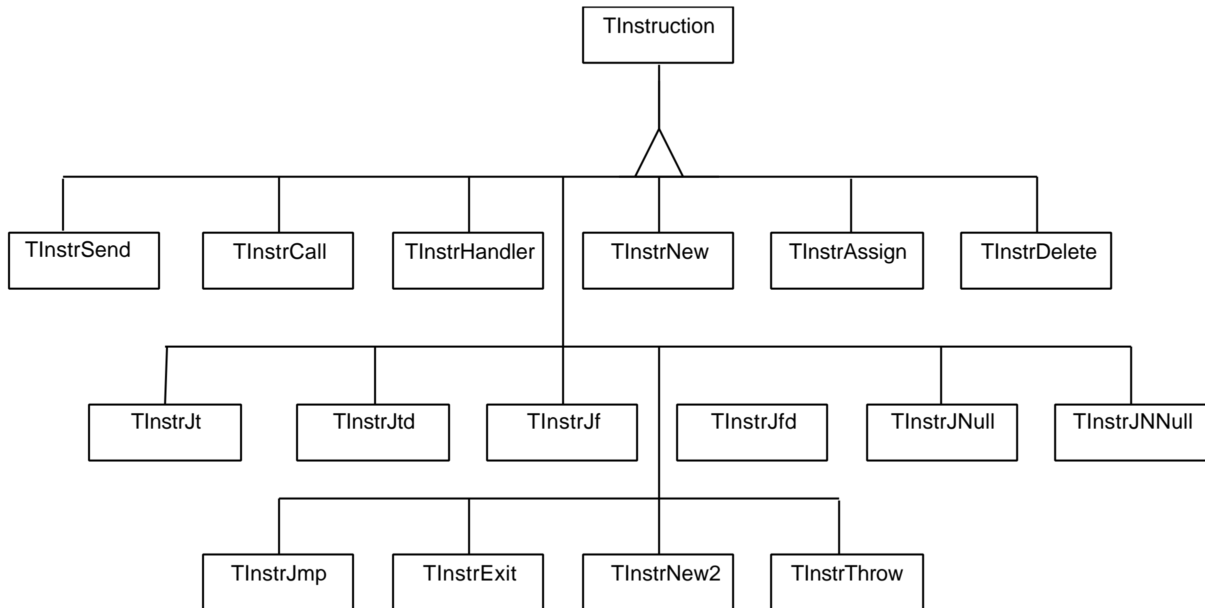


Figura 17.5 Jerarquía de TMethod para los métodos con implementación primitiva.

Este tipo de implementación con este grado de encapsulamiento permite añadir fácilmente clases derivadas sin que el resto del simulador quede afectado. Esto implica que es muy sencillo añadir nuevas instrucciones a la máquina, nuevas clases y métodos primitivos, etc. Basta con añadir una nueva subclase que derive de la clase abstracta que implemente la nueva instrucción, clase, etc. y automáticamente el resto del sistema la usa sin necesidad de otras modificaciones.

3 Entorno de Computación en tiempo de ejecución definido por la máquina

El entorno de computación de la máquina viene determinado fundamentalmente por las instrucciones primitivas que la máquina ofrece para ejecutar los métodos de los objetos, por los objetos que representan la ejecución de los métodos y sus relaciones y por la lógica de la máquina que describe el flujo de control, representado mediante invocación de métodos de esos objetos primitivos, de unos a otros para llevar a cabo la ejecución efectiva del método solicitado.

La arquitectura aquí presentada cobra vida a la hora de ejecutar los programas escritos en el **lenguaje Carbayón** (lenguaje de la máquina) que el usuario proporciona. La máquina abstracta interpreta los programas ejecutando las instrucciones una a una, según se señale en el programa.

La ejecución de un programa se apoya, además de en la ejecución de las instrucciones primitivas que la máquina proporcione, en la definición de objetos y la

[1] Apunta a la siguiente instrucción a ejecutar dentro del método

ejecución de sus métodos. Por tanto, para lograr una completa visión del funcionamiento de la máquina, es necesario dar una visión dinámica de la misma, o lo que es lo mismo, una perspectiva de la misma en tiempo de ejecución, describiendo cómo reacciona la máquina ante la ejecución de una instrucción a petición de un programa de usuario, qué objetos utiliza para describir esa ejecución, qué objetos representan el flujo de control, qué relaciones existen entre ellos, cuándo y por quién son invocados sus métodos y cómo se retornar los resultados en la cadena de llamadas.

3.1 Representación del entorno en tiempo de ejecución

3.1.1 Representación de los objetos en tiempo de ejecución

En tiempo de ejecución, una instancia de un objeto tiene una correspondencia directa con una instancia de la clase abstracta `Tinstance` de la máquina, que describe una instancia de un objeto en tiempo de ejecución utilizando un formato idéntico para todos los objetos, independientemente de su funcionalidad o de si se trata de un objeto primitivo o definido por el usuario.

En la siguiente figura se puede apreciar las relaciones con otros elementos de la máquina.

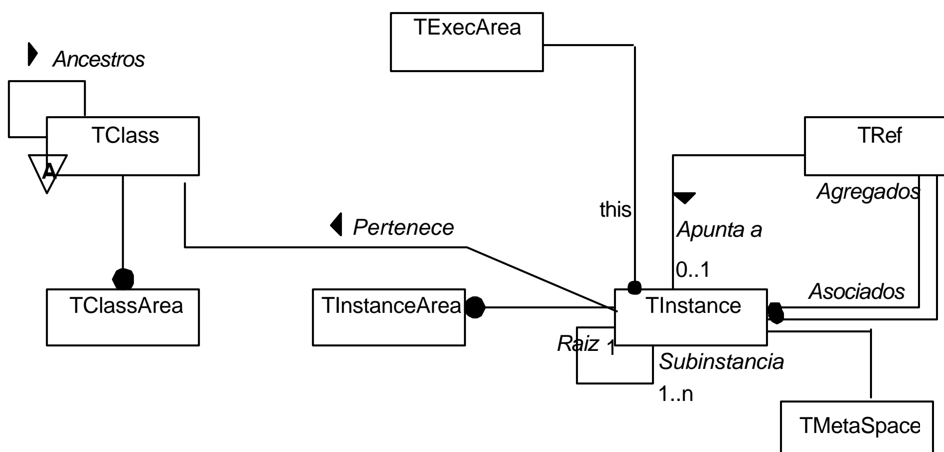


Figura 17.6 Diagrama de clases que muestra las relaciones de las Instancias con el resto de los elementos de la máquina

Cada instancia pertenece a una clase y tiene cero o más referencias a instancias agregadas o asociadas. Todas las instancias están almacenadas en el Área de Instancias (TInstanceArea) que representa al área de instancias de la máquina abstracta. Además, la forma de referirse a una instancia es a través de una referencia.

Por otra parte, cada instancia tiene asociada un Área de Ejecución (TExecArea) que almacena los objetos internos de la máquina que representen la ejecución en curso de los distintos métodos en este objeto.

Por último, cada instancia tiene asociado un objeto TMetaSpace que almacena las referencias a los meta objetos que constituyen el entorno en tiempo de ejecución de la instancia.

3.1.2 Representación de la Computación

La versión de la máquina abstracta Carbayonia que aquí se describe, es una máquina multihilo con soporte explícito para la concurrencia.

La ejecución de métodos de objetos, solicitada mediante la ejecución de las instrucciones primitivas `call` o `Send`, desencadena una cascada de llamadas a métodos a los objetos internos que la máquina define para describir la ejecución de un método en un objeto, como se verá en el apartado 4.2.

Fundamentalmente, la máquina representa la ejecución de métodos mediante objetos internos denominados **hilos de ejecución**. Son objetos de la máquina abstracta que representan cadenas de invocación de métodos en un mismo objeto, y por tanto, secuencias de ejecución de las instrucciones de esos métodos, en tiempo de ejecución.

Cada hilo almacena también la información necesaria para permitir navegar por la cadena de llamadas/respuesta, construyendo una nueva entidad, que no tiene representación explícita y que representa la cadena de invocación a métodos en distintos objetos de la máquina.

La máquina abstracta crea hilos de ejecución cuando se envía un mensaje y se destruyen cuando se le acaba la pila de llamadas o bien se produce una excepción que no es atrapada.

Aunque en principio se afirme que los hilos son objetos internos, no visibles al exterior, más adelante se mostrará cómo se exponen al exterior no sólo su existencia sino también su estructura[Riv96], lo que permitirá consultar e incluso modificar la ejecución. Esta exposición es imprescindible para implementar la planificación a nivel de usuario.

3.1.3 Implantación

La implantación de los mismos es similar a la de la Máquina-p [NAJ+76]. Cada objeto hilo (TThread) tiene una pila asociada compuesta por contextos(TContext).

Un **contexto** representa el contexto de ejecución de un método o entorno en que se ejecuta un determinado método, de tal forma que las instrucciones del método actuarán sobre la información almacenada en el contexto.

Un contexto proporciona un área de almacenamiento de los elementos dinámicos de ejecución de un método:

- un método, sobre el que se ejecuta.
- una instancia, sobre la que se invocó el método.
- cero o mas referencias que apuntarán a los parámetros y la referencias locales.
- cero o mas referencias que apuntarán a las instancias locales.
- Una referencia en la cual es donde deberá dejar el valor de retorno o la excepción al salir del contexto (rr o exc).

Al finalizar el método o cuando se produce una excepción y no hay ningún handler posterior a él en la pila, se elimina el contexto en cuyo caso se liberan todas sus instancias locales.

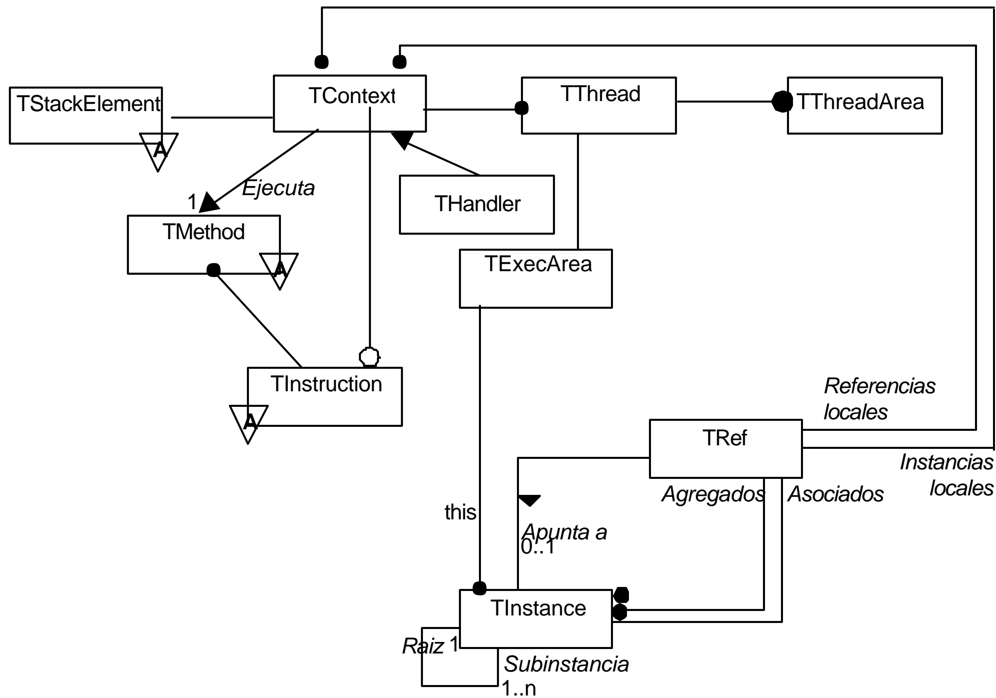


Figura 17.7 Diagrama de clases que muestra los objetos involucrados en la ejecución

4 Ciclo de Ejecución de Instrucciones

Dado que el propio diseño interno de la máquina abstracta se ha realizado aplicando los principios de OO (ver figura 17.1), la ejecución de un método supone una cascada de llamadas a distintos objetos de la máquina.

4.1 Ejecución de una Instrucción

Cuando la máquina, siguiendo el ciclo de ejecución, decide ejecutar la siguiente instrucción del objeto hilo actual, invoca en este el método trace, que ejecuta la siguiente instrucción del contexto actual.

Para ello, se recupera el MC actual (method counter) que indica cuál es el número de la instrucción siguiente a ejecutar dentro del método actual, e invoca el método trace en el objeto método, para que ejecute dicha instrucción.

El objeto método se limita a retransmitir la petición al objeto instrucción que ocupe dicho lugar mediante la función exec. Este objeto será una instancia de una de las clases derivadas de TInstruction que corresponde con una de las instrucciones de la máquina abstracta por lo que, a partir de este momento, la secuencia de llamadas del escenario depende de cuál fuera dicha instrucción (ya que debido al polimorfismo la llamada a exec puede dar lugar a la ejecución de distintos métodos).

El escenario general de la ejecución de una instrucción sería el que se muestra en la Figura 17.8.

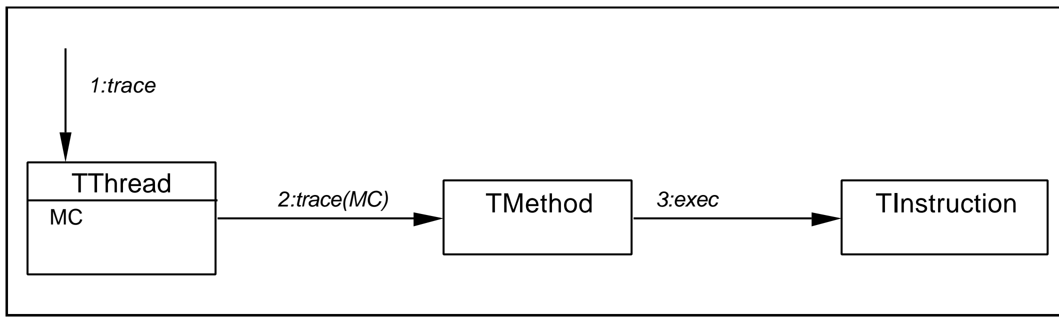


Figura 17.8 Ejecución de una instrucción

4.2 Invocación de Métodos

La invocación de métodos o paso de mensajes es, además del único mecanismo de comunicación entre objetos, el medio de introducir la concurrencia en la máquina. Dependiendo de si la máquina decide o no crear un nuevo hilo para ejecutar un método, el nivel de concurrencia aumenta o se mantiene.

La ejecución de métodos en este prototipo se desencadena tras la ejecución, por parte de un hilo, de las instrucciones Call o Send, únicas instrucciones para la ejecución de métodos. Suponen la ejecución del conjunto de instrucciones que componen el método invocado.

Concurrencia

Dado que la concurrencia se mantiene, fundamentalmente, en función de la creación o no de nuevos hilos ante la invocación de métodos, la definición de la semántica de ambas instrucciones determinará completamente el nivel de concurrencia permitido en la máquina.

4.2.1 Ejecución de Métodos

La invocación de un método comienza cuando llega el mensaje exec a una instancia de TInstrCall o TInstrSend, encargadas de realizar las llamadas a los métodos.

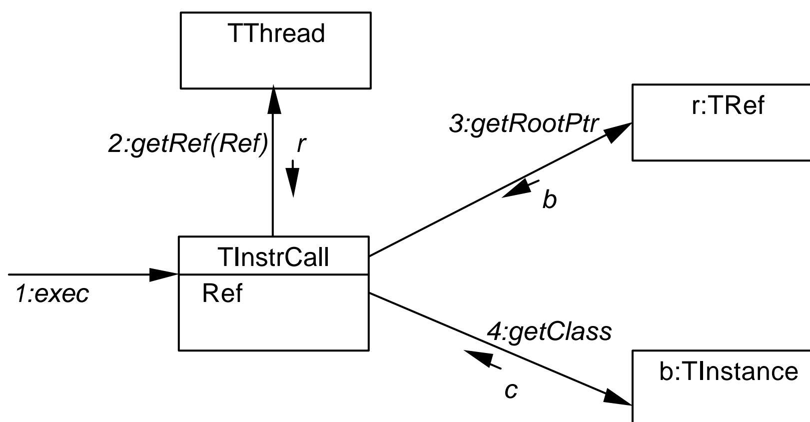


Figura 17.9 Ejecución de una instrucción Call o Send

Cuando la TInstrCall o la TInstrSend reciben la petición de ejecutarse solicitan al thread que consiga acceso a la referencia sobre la que se ha realizado la invocación para identificar la clase a la que pertenece.

Dependiendo de si se trata de un objeto primitivo o si, por el contrario, es un objeto de usuario, se producirán dos subescenarios.

Ejecución de un método en un objeto con implantación primitiva

Si se trata de un objeto primitivo, no se permite ningún tipo de personalización en la invocación a métodos o control de la concurrencia por parte del usuario. Además, los métodos se ejecutan síncronamente, dentro del mismo hilo que invoca al método, por lo que no existe planificación de ningún tipo.

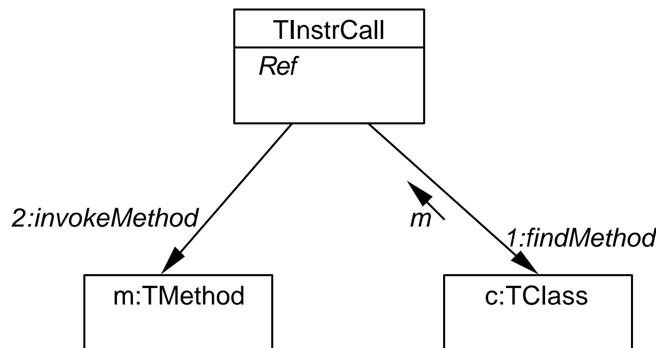


Figura 17.10 Invocación de un método primitivo

Como escenario final, y como representante de todos los métodos primitivos, se muestra ahora el escenario en el caso de que el método hubiese sido un método primitivo, concretamente la suma de números enteros (integer::add).

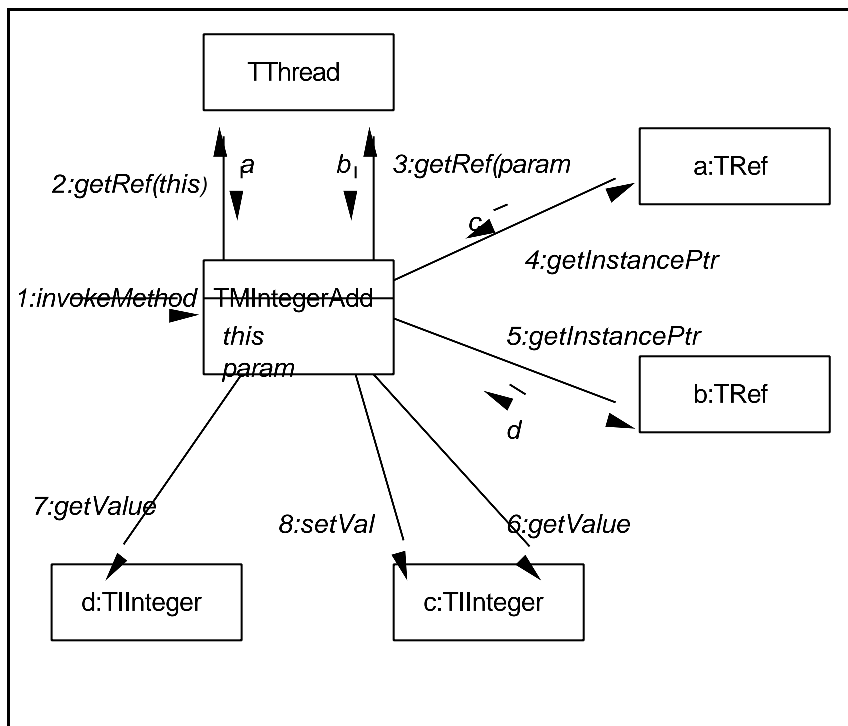


Figura 17.11 Ejecución de un método primitivo

El método de suma recibe la notificación de que le han invocado mediante invokeMethod.

Solicita al thread sobre el que se está ejecutando el método, que obtenga acceso a la referencia sobre la que se ha invocado el método y la que se ha enviado como parámetro (pasos 2 y 3). Una vez localizadas las referencias se le solicita las direcciones de los objetos a los que apuntan (pasos 4 y 5).

Una vez que se tiene acceso directo a las instancias de tipo TIInteger se les pide el valor de cada una y se establece en la primera el resultado de sumar ambas.

Nótese que no se realiza ningún cambio en el contexto del thread, por lo que la siguiente instrucción a ejecutarse es la que sigue a la invocación del método Add. Se puede considerar como que el método crea un contexto sobre el que se ejecuta y al finalizar su ejecución sale de él.

Ejecución de un método en un objeto de usuario

Si se trata de un objeto de usuario, sí se permite personalizar distintos aspectos del entorno de ejecución, entre ellos, el paso de mensajes. El método sería entonces un TUserMethod.

A continuación se muestra cómo se resolvería en el caso de una instrucción Call. El tratamiento para Send sería idéntico. La diferencia se establece posteriormente a este estadio de la ejecución.

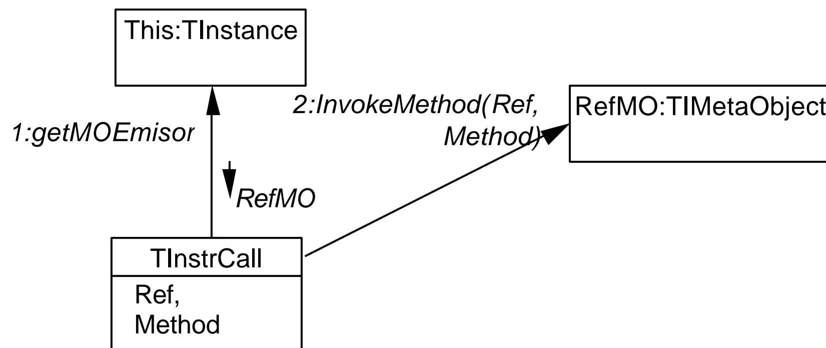


Figura 17.12 Invocación de un método en un objeto definido por el usuario

El resto se saldría ya del ámbito de la máquina y entraría de lleno en el del Sistema Operativo, por lo que se verá más adelante.

Sea lo que sea lo que el meta objeto correspondiente ejecute, al final, debe invocar el método invokemethod en el objeto método correspondiente. Esta situación se describe en el siguiente diagrama de objetos.

El primero de ellos es si el método corresponde a una instancia de TUserMethod (definido por el usuario). Entonces la situación corresponde al siguiente diagrama.

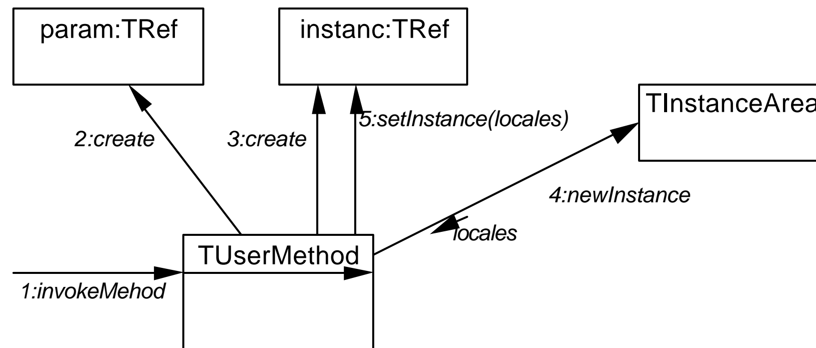


Figura 17.13 Ejecución de un método de usuario

Cuando el TUserMethod recibe el mensaje crea todas las referencias que le hagan falta para representar sus argumentos, referencias e instancias locales. En el caso de estas últimas además creará las instancias a las que apunten solicitándoselo al área de instancias (pasos 4 y 5).

4.2.2 Retorno de Resultados

El retorno de resultados lo realiza la propia máquina abstracta Carbayonia, con lo que se evita tener que introducir primitivas de comunicación explícitas tipo wait y reply para el retorno de los resultados, consiguiendo así una máquina tipo RISC más sencilla.

La máquina Carbayonia lleva a cabo el retorno de resultados desapilando un contexto del objeto hilo, que pasa a ejecutarse en el contexto desde el que se realizó la invocación al método, y desbloqueando así al objeto que se hallaba esperando la respuesta.

En caso de que sea el último, pasa a ejecutarse en el contexto superior del hilo que provocó la creación del hilo que termina.

El modelo de devolución de resultados síncrono se basa en que, cada contexto del hilo, tiene una referencia especial, la rr (return reference) donde la máquina guarda el resultado esperado.

La máquina se encarga de poner en rr el valor correspondiente, poner al objeto destino listo para ejecutar.

Cuando el objeto destino reanude su ejecución, la máquina mira si es un retorno normal o una excepción. De esta forma, en este modelo la gestión de las excepciones se lleva a cabo íntegramente en la máquina y de forma secuencial y síncrona.

El modelo de devolución de resultados asíncrono siguiendo la semántica wait-by-necessity se basa en que cada referencia puede estar instanciada o no. En el caso de que no lo esté, la máquina abstracta bloquea al hilo que invoca un método en la misma. Si está instanciada, se continúa la ejecución como hasta ahora.

4.3 Excepciones

El tratamiento de excepciones está incorporado a bajo nivel en Carbayonia, lo que, por un lado, facilita la tarea de depuración y mantenimiento de las aplicaciones, gracias a la información que se guarda en las áreas de clases, referencias e instancias. Y por otro

lado, promueve su utilización en los programas de usuario y por tanto la generación de un código mas robusto por parte de éstos.

4.3.1 Implementación

Para implementar las excepciones, se hace que la pila de un hilo pueda entremezclar elementos para representar los manejadores de excepciones (mediante una clase abstracta TStackElement de la que derivan los contextos TContext y manejadores THandler).

En vez de dar soporte directamente a una estructura compleja tipo try/catch, la opción que finalmente se eligió fue incluir dos instrucciones que den soporte a su construcción.

Estas dos instrucciones son handler y throw. La segunda no tiene operandos, mientras que el formato de la primera es:

Handler <etiqueta>

La misión de dicha instrucción es indicar la dirección donde se debe continuar el flujo de ejecución en caso de que ocurra una excepción. La dirección será aquella que corresponda a la posición de la etiqueta en el código fuente.

Cuando se ejecute un throw, el objeto lanzado estará referenciado por la referencia exc del contexto actual. La ejecución del programa continuará en la dirección del último handler ejecutado. En caso de que no haya ninguno, la ejecución se dará por finalizada. Los distintos handler se van apilando de manera que tienen prioridad los últimos que se hayan ejecutado.

```
      handler atrapa1
      handler atrapa2

      throw
      exit // No se ejecutará

atrapa1:
      // No vendrá aquí
atrapa2:
      // Aquí continua
```

En el ejemplo anterior se activan dos handlers y a continuación se produce una excepción. El control pasará a la etiqueta atrapa2. En caso de que una vez ya en esta posición se produjese otra excepción es cuando el control pasaría a atrapa1.

La cesión de control a un handler se realiza aunque para ello haya que abandonar el método actual. Cuando se lanza una excepción en un método que no ha definido un handler, se retrocede en la secuencia de llamadas que llevó hasta la situación actual intentando encontrar algún método de lo recorridos que hubiese declarado un handler. Cada método que deba abandonarse por causa de una excepción libera previamente los objetos declarados en la sección instances del método.

```
Metodo1()
Code
      handler atrapa;
      this.metodo2();

      exit; // No se ejecutará
```

```
atrapa:
      // Vendrá aqui

EndCode

Metodo2()
Instances
      var:integer;
Code
      throw;
EndCode
```

En el ejemplo anterior el método1 llama al método2. Dentro de este se produce una excepción. Dado que el último handler encontrado esta fuera del método actual se liberan las instancias locales (var) y se sale del método. Una vez de vuelta en metodo1 el control pasa a la posición de la etiqueta atrapa.

Los handler no solo se extraen cuando se produce una excepción, sino que también se descartan cuando se sale del método en el que fueron declarados sin que se produjese una excepción.

```
Metodo1()
Code
      this.metodo2();

      throw; // No irá a atrapa

EndCode

Metodo2()
Instances
      var:integer;
Code
      handler atrapa;
      exit;

atrapa:
      // No vendrá aqui
EndCode
```

En este caso anterior el metodo1 invoca al método2 y este establece un handler. Sin embargo al llegar la instrucción exit esta descarta todos los handler declarados desde la entrada al método, por lo que al volver al método1 y producirse la excepción el control no pasa a la etiqueta atrapa del metodo2. En este caso el control pasaría a aquel método que llamó a metodo1 que hubiese declarado un handler antes de hacerlo.

5 Reflectividad Estructural en la máquina

Dentro del prototipo, una parte muy importante fue el diseño del mismo para conseguir dotar a la máquina de una arquitectura reflectiva.

Por una parte esto se logra haciendo que la instrucción de invocación de método se comporte como se describió arriba.

Por otra parte, es necesario exponer los elementos de implantación de la máquina, lo que supone la creación de una jerarquía de clases alternativa que permita al sistema acceder a su propia implantación interna y modificar su comportamiento.

5.1 Descripción de la Jerarquía de Clases para la Reflectividad Estructural

La implantación de una máquina dotada de reflectividad estructural se lleva a cabo mediante la exposición de los objetos del nivel base.

Para ello, se definen un conjunto de clases que exponen los aspectos de implantación de esos objetos, tal como se explicó en el Capítulo XV y se muestra en el Anexo C.

En la figura 17.14 se ofrece una visión de la jerarquía de clases.

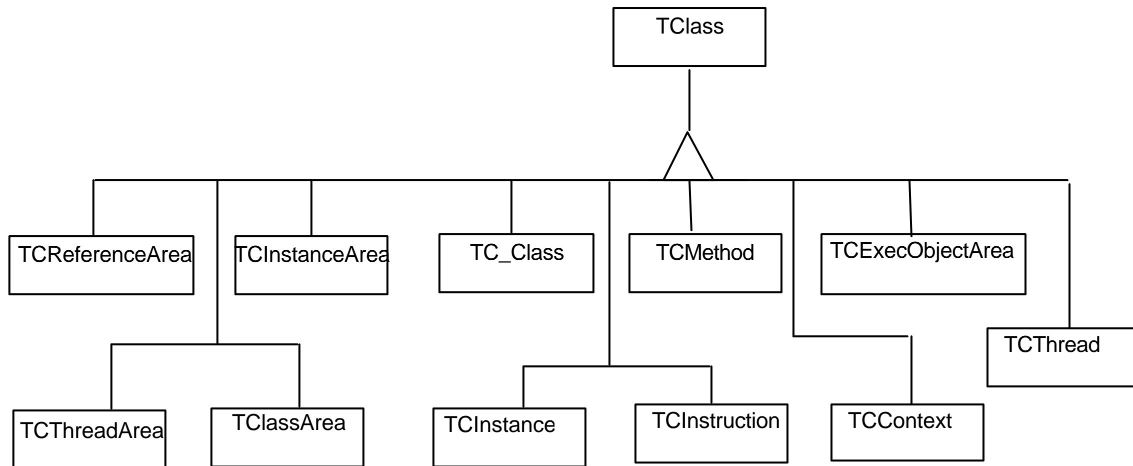


Figura 17.14 Jerarquía de Clases para la Reflectividad Estructural en Carbayonia

Este conjunto de clases, que se corresponde totalmente con la descripción de alto nivel que se ofrece en el Anexo C, permiten al sistema acceder a los datos en tiempo de ejecución, realizando introspección e incluso intercesión sobre él.

Esto último es fundamental ya que, dado que el meta-sistema no es más que un reflejo del sistema, la actuación sobre él se traduce en modificaciones similares en el sistema base reflejado.

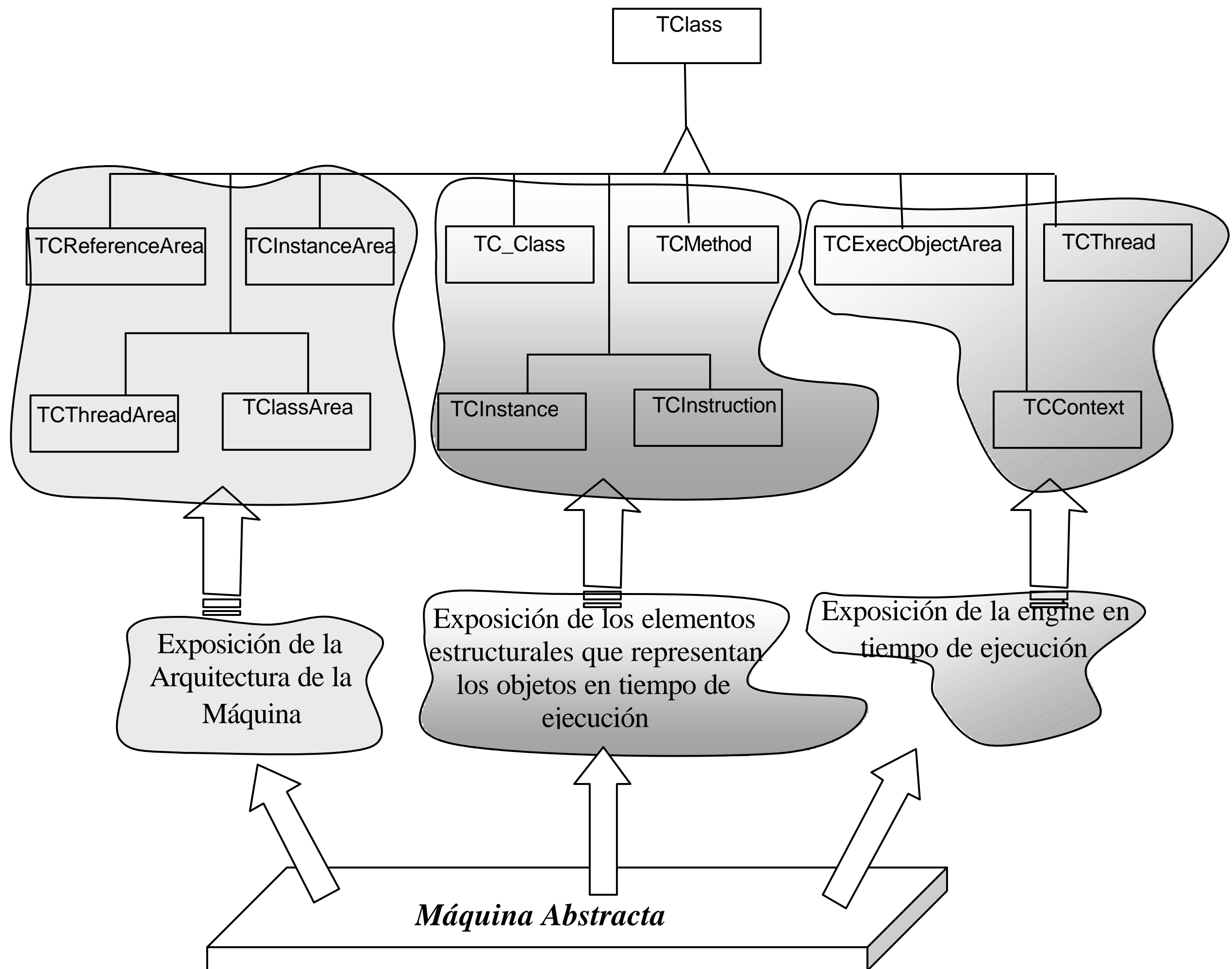


Figura 17.15 Exposición de distintos componentes de la máquina abstracta a través de la Jerarquía de Clases para la Reflectividad Estructural en Carbayonia

Así, la modificación de un método, supone la modificación del comportamiento de la clase ya que, a partir de ese momento, actuará con el método modificado.

En la figura 17.16 se muestra gráficamente la forma de cosificar los objetos del sistema base a través de los objetos del módulo de reflectividad estructural.

6 Sistema Operativo: Reflectividad del Comportamiento

6.1 Descripción de la Jerarquía de Clases para la Reflectividad del Comportamiento

Además de exponer la estructura de los objetos del nivel base también se exponen partes del comportamiento de la máquina, mediante una jerarquía de clases similar a la anterior y que permite la modificación del comportamiento de la misma, adaptándolo a cada aplicación.

Los elementos del comportamiento de la máquina que se exponen son la invocación de métodos, el modelo de objetos para la concurrencia y la planificación, tal como se indicó en los capítulos XIV y XVI.

A diferencia de la jerarquía anterior, esta nueva jerarquía se corresponde con la funcionalidad del Sistema Operativo.

Sin embargo, la raíz de la jerarquía está definida en la máquina, fundamentalmente, por cuestiones de eficiencia.

Esta jerarquía es ampliable, mediante herencia, añadiendo nuevas clases, tal y como se presenta en el Anexo C.

La finalidad del Sistema Operativo es exponer mediante objetos instancia de sus clases, algunas partes de la implantación de la máquina.

Para ello, se instancian objetos de las clases presentadas que representan distintas partes del comportamiento de la máquina. Estos objetos de alto nivel, son modificables, lo que permite la adaptación del comportamiento de la máquina.

Estos objetos son instancias de clases, formando una jerarquía que contiene las clases correspondientes a las que se detallan en el Anexo C.

En la figura 17.17 se ofrece una visión de esta jerarquía de clases.

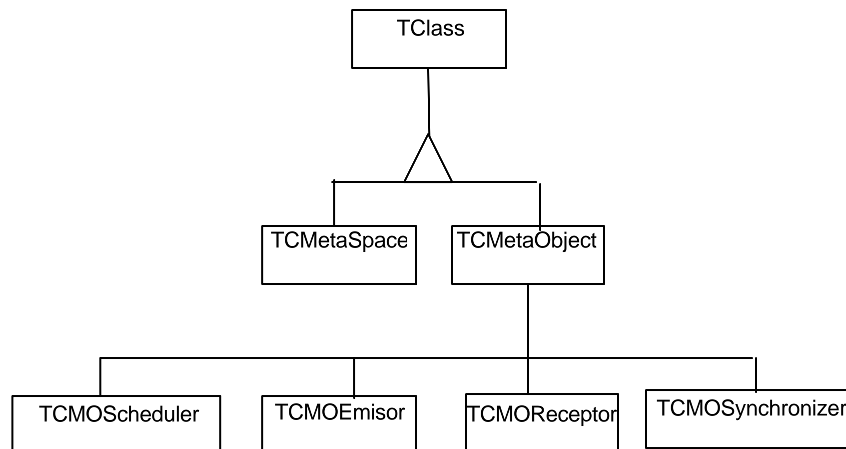


Figura 17.17 Jerarquía de Clases para la Reflectividad del Comportamiento en Carbayonia

Este conjunto de clases permiten al sistema acceder a los datos en tiempo de ejecución acerca del comportamiento particular que la máquina ofrece para él. La modificación de este comportamiento se lleva a cabo a través de la ampliación de la jerarquía mostrada en el Anexo C y de la modificación del objeto MetaSpace del objeto base.

La modificación de la jerarquía puede llevarse a cabo tanto añadiendo clases en tiempo de compilación como en tiempo de ejecución, gracias a la reflectividad estructural que expone el objeto clase de la máquina abstracta y la exposición del área de clases de la máquina.

6.1.1 Comunicación en el Sistema Operativo

La máquina abstracta, como se vio en la figura 17.5 implementa instrucciones Call y Send, que se comportan, no sólo como primitivas de comunicación a bajo nivel para los objetos del meta espacio, sino como traps al Sistema Operativo, ya que pasan el control a los objetos del Sistema Operativo en el meta espacio.

Una de las partes de la máquina abstracta sobreescrita a alto nivel por los objetos del Sistema Operativo es la comunicación entre objetos del nivel base, es decir, la invocación de métodos.

Como se vio en la figura 17.12, ambas instrucciones invocan el método invokeMethod en el objeto MOEmisor.

Lo que ocurre a partir de este momento, entra completamente dentro del simulador del Sistema Operativo.

A continuación se muestra el diagrama de colaboración de objetos para conseguir simular a alto nivel el comportamiento de la máquina abstracta en este sentido.

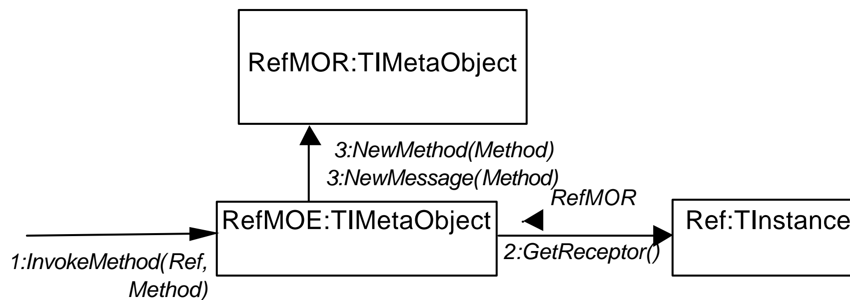


Figura 17.18 Relación entre meta objetos para la comunicación entre objetos

Es necesario notar aquí que, tanto el Meta objeto Emisor (RefMOE) como el meta objeto receptor (RefMOR) pueden realizar muchas más acciones antes y después de las que se señalan en la figura 17.18. Sin embargo, por simplicidad, se opta por mostrar únicamente las relaciones imprescindibles entre ellos.

6.1.2 Modelo de Objetos para la Concurrencia en el Sistema Operativo

Una vez que el Receptor de un objeto recibe un mensaje, todo se lleva a cabo en el lado del receptor.

La ejecución del nuevo método viene condicionada al modelo de concurrencia definido y al estado actual del objeto. Una versión simplificada es la que se muestra en la figura 17.19.

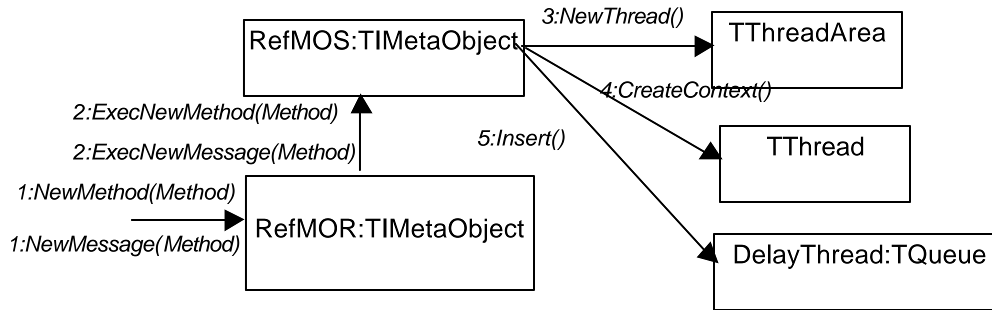


Figura 17.19 Definición del modelo de objetos para la Concurrencia en el meta espacio.

Si el método invocado es actualmente ejecutable se crea el hilo correspondiente y se continúa la ejecución saltándose el paso 5, que supone la inserción del mismo en una cola de hilos a la espera, por ser el actual un momento inadecuado para la ejecución. A partir de este momento, dependiendo de si la invocación fue síncrona (RefMOR→NewMethod) o asíncrona (RefMOR→NewMethod) las acciones transcurren por diferente cauce.

El nuevo objeto hilo creado, representa la ejecución del método solicitado y conoce la identidad de su antecesor en la cadena de llamadas/respuesta de tal forma que es sencillo retornar hacia atrás o moverse hacia delante desde cualquier punto de la mencionada cadena de invocaciones.

Si la invocación fue síncrona, se salta la planificación y se ejecuta lo siguiente de forma síncrona, de tal modo que no se devuelve el control al objeto emisor hasta que termine.

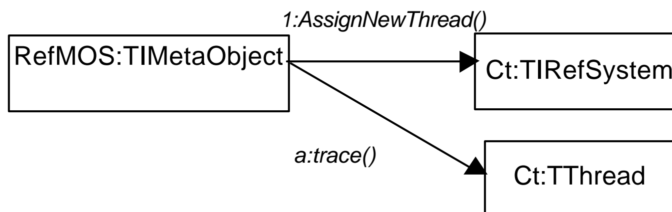


Figura 17.20 Finalización de la invocación de un método

Es decir, se asigna como nuevo hilo a ejecutar el hilo recientemente creado y se le cede la CPU directamente, al invocar el método trace del hilo referenciado por la referencia del sistema ct.

6.1.3 Planificación en el Sistema Operativo

En caso de que la invocación fuese asíncrona, o si el método se bloquea porque no puede ejecutarse en ese momento, o por alguna otra razón, o bien si se termina el tiempo de ejecución del método en curso, entra en juego el objeto planificador. Por simplicidad se mostrarán aquí sólo algunos de los escenarios de la planificación.

En primer lugar, en la figura 17.21 se muestra cómo el MetaObjeto Sincronizador puede invocar el método EnqueueNewThread, para dar a conocer al planificador la existencia de un nuevo hilo a tener en cuenta.

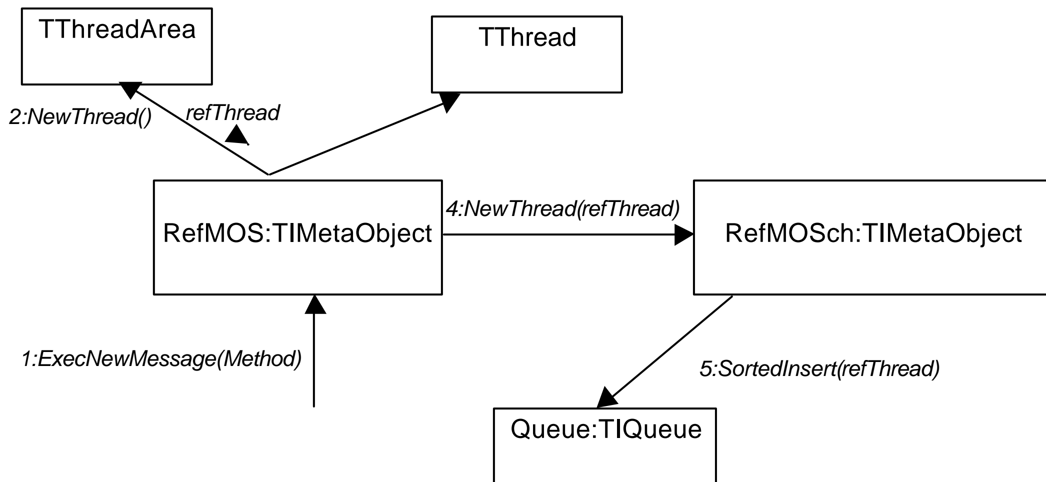


Figura 17.21 Inserción de un nuevo hilo en el Planificador

Por otra parte, pueden ser varias las causas que provoquen que un planificador tenga que elegir un nuevo hilo para ejecutar, desde el final del quantum del hilo actual, hasta el bloqueo del mismo en un semáforo, etc.

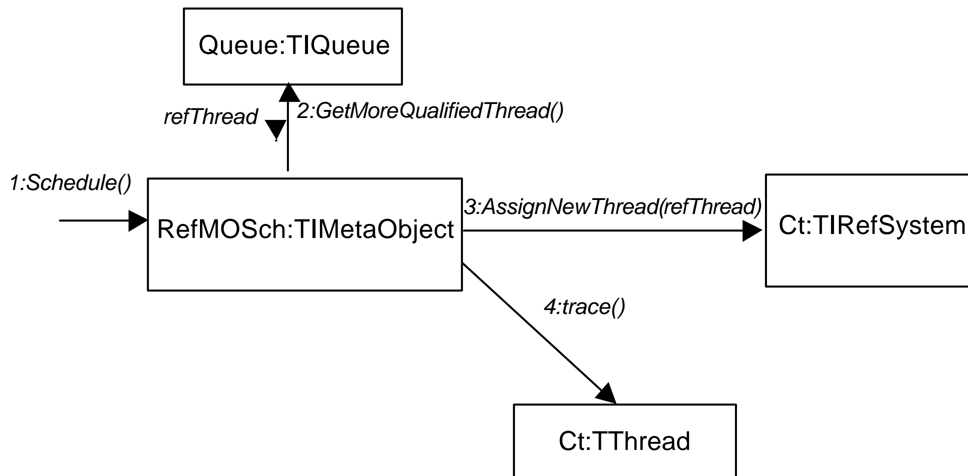


Figura 17.22 Planificación de un nuevo hilo

Capítulo XVIII

Conclusiones del Trabajo y Resultados más Destacables

Fundamentalmente, los objetivos de esta tesis eran, inicialmente, dos. En primer lugar demostrar que la construcción de un SIOO tomando como base la combinación reflectiva de una Máquina Abstracta Orientada a Objetos y un Sistema Operativo Orientado a Objetos, es factible. En segundo lugar, integrar la computación en el entorno OO sin fisuras ni abstracciones ajenas al objeto.

En el curso del desarrollo de la misma, se han ido tomando decisiones de diseño extrapolables a otros sistemas. Estas decisiones se presentan aquí en el apartado "Resultados más Destacables".

Antes de nada, el capítulo comienza con una breve recapitulación de los contenidos más reseñables del documento.

1 Breve Resumen del Documento

1.1 Planteamiento del problema: Construcción de un Sistema de Soporte de Objetos

Los inicios de esta tesis se basan en la exposición o planteamiento de dos graves problemas que aquejan el diseño y construcción de sistemas de soporte de objetos aplicando la Tecnología de Orientación a Objetos.

El Sistema Orientado a Objetos desde fuera: Sistema Realmente Orientado a Objetos

Por un lado, se trata de resolver el problema del diseño y construcción de un entorno realmente orientado a objetos.

Muchos sistemas actuales propugnan su identidad dentro del grupo de los Sistemas Orientados a Objetos. Sin embargo, el soporte que ofrecen a la abstracción de objeto es, cuando menos, parcial y limitado.

Muchos de ellos sólo aplican la OO en tiempo de diseño. Esto implica no pocas ventajas, aquellas que se derivan de la construcción de cualquier software aplicando Tecnologías OO. Mucho más si el software es grande y complejo como puede resultar un SIOO.

Sin embargo, en tiempo de ejecución, las abstracciones que ofrece son las tradicionales que, como se vio en los Capítulos I y III, distan mucho de encajar adecuadamente con el objeto.

Esta integración de abstracciones tradicionales para soportar objetos provoca la aparición de problemas de interoperabilidad, poca extensibilidad del entorno y degradación del rendimiento, que se comentaron ampliamente en el capítulo I.

El Sistema Orientado a Objetos desde dentro: Integración Sin Fisuras de la Computación en el Sistema Orientado a Objetos

El segundo problema al que se intenta dar solución en esta tesis, es la integración en el Sistema Orientado a Objetos de la computación de objetos.

El objetivo fundamental a la hora de diseñar el sistema de computación del SOOO, es integrar el concepto de objeto y la computación del mismo, manteniéndonos fieles a los principios de abstracción única y uniforme y homogeneidad en el modelo.

Esto supone, por un lado, que no existen nuevas abstracciones, como el proceso, sino que es necesario que la abstracción de objeto englobe la computación del entorno.

Por otro lado, la abstracción de objeto debe estar diseñada con unas características básicas a las que, cualquier objeto que exista en el sistema en tiempo de ejecución, debe mantenerse fiel.

Se define pues un modelo único de objetos, también para la computación.

El capítulo III estudia y posteriormente rechaza los modelos tradicionales de computación, basados, todos ellos, en una abstracción adicional, proceso, que encaja mal con el entorno OO que se pretende diseñar.

Posteriormente, en los Capítulos VII, VIII, IX y X se afronta el problema decidiendo adoptar un modelo de objetos activo.

1.2 Diseño de un entorno de Computación OO

El resto de esta tesis se centra en la búsqueda de la solución que se puede resumir como la construcción de un entorno de computación OO en el que se dé soporte a una única abstracción, el objeto como entidad con semántica propia y no un mero contenedor de datos.

1.2.1 Diseño de un SIOO

Como parte de la solución, el Capítulo V propugna la construcción de un SIOO como solución global a los problemas de interoperabilidad descritos en la primera parte de la tesis, mostrando además, la anatomía del SIOO propuesto en esta tesis, compuesto por una Máquina Abstracta Orientada a Objetos (MAOO) sobre el que se define un Sistema Operativo Orientado a Objetos (SOOO).

En este capítulo, también se definen los requisitos básicos del SIOO y se detallan las características del modelo de objetos adoptado.

Requisitos Básicos

Los requisitos básicos giran en torno a la simplicidad.

- Uniformidad. Se trata de dar soporte a una única abstracción, el objeto, independientemente de su granularidad o el lenguaje del que procede. Esto implica

que no existen diferentes categorías de objetos en el sistema, como podría ser el objeto de datos y el objeto proceso.

- Homogeneidad. Es un punto crucial en el diseño del sistema. La abstracción de objeto que se ofrece debe ser homogénea, lo cual implica que todos los objetos del sistema deben tener las mismas propiedades y el mismo tratamiento. No pueden definirse objetos especiales e incluso los objetos del sistema no son diferentes a cualquier otro objeto definido por el usuario. El soporte y funcionalidad que se ofrece, es el mismo para todos ellos.
- Simplicidad. El sistema debe mantenerse lo más sencillo posible.

Modelo de Objetos

En este capítulo, también se detallan las características del modelo de objetos adoptado.

Se trata de que sea lo más estándar posible, lo que supone adoptar características como encapsulación, jerarquía o polimorfismo, entre otras, aunque añadiendo otras características que, si bien no suelen mencionarse como parte del modelo de objetos, sí parecen fundamentales, como la concurrencia.

En el Capítulo VI se analizan los distintos grados de soporte a objetos, y las distintas posibilidades que se presentan de construir un sistema Operativo orientado a objetos que dé soporte a objetos.

Para mantener la homogeneidad con el resto del sistema y tras analizar distintas posibilidades, la conclusión que se presenta es que el Sistema Operativo, en tiempo de ejecución, debe estar compuesto por un conjunto de objetos, homogéneos con el resto de los objetos. De esta forma, la solicitud de servicios se hará mediante la invocación de métodos en estos objetos.

Este capítulo se enfoca ya hacia uno de los mayores aportes de esta tesis, como es la necesaria integración, de forma homogénea y flexible, del Sistema Operativo y la máquina Abstracta.

En el Capítulo XI se analiza más a fondo la necesaria combinación de ambos elementos, que constituyen la base del SIOO y se introduce la Reflectividad como medio para combinar de forma uniforme ambas capas.

El Capítulo XII ofrece un estudio más profundo de la Reflectividad.

1.2.2 Integración de la Computación en el Entorno OO

Los capítulos VII, VIII, IX y X se dedican a la solución del segundo problema, la integración sin fisuras de la computación en el entorno integral.

El Capítulo VIII es la exposición de los requisitos funcionales que se espera que cumpla el entorno de computación OO. No se distinguen mucho de los requisitos funcionales que se espera de otros entorno más tradicionales.

La elección de una abstracción adecuada es, sin duda el punto más peliagudo, habida cuenta de que, ya en los Capítulos V y VI, se estableció como objetivo irrenunciable la abstracción única en el objeto.

La comunicación entre objetos, la ejecución concurrente de los métodos de un mismo o distintos objetos o la planificación de las actividades concurrentes son los requisitos funcionales más habituales y por ello, los más conocidos.

A ellos hay que unir un requisito menos habitual, el mecanismo de excepciones como medio para comunicar al entorno la ocurrencia de un evento anormal.

Por su parte, en el Capítulo IX se establecen los requisitos no funcionales, aquellos parámetros según los cuales se medirá la bondad con la que el entorno de computación OO consigue cumplir los requisitos funcionales anteriormente expuestos. Estos requisitos son una extensión de los requisitos expuestos en general para el SIOO en el Capítulo V.

Por un lado, un modelo único de objetos también para la computación, la homogeneidad en torno al objeto y la necesaria flexibilidad de los mecanismos de gestión de la computación concurrente, que permitan eliminar el habitual corsé que el Sistema Operativo supone para las aplicaciones.

Por último, en el Capítulo X se establece el modelo de objetos para la computación. Este modelo de objetos se basa en integrar dentro del objeto la abstracción de ejecución de sus métodos para lo cual la máquina abstracta deberá ofrecer un soporte mucho más completo a la abstracción de objeto.

Se ofrece, ya desde los niveles más básicos del SIOO, una abstracción de objeto como mecanismo de computación autónomo, que se comunica mediante el paso de mensajes.

1.2.3 Reflectividad en el Sistema

La propuesta, como solución a los problemas de soporte de objetos, de un SIOO compuesto por una máquina abstracta OO y un sistema operativo OO, supone la necesaria integración de ambos mediante algún tipo de arquitectura. Esto se discute en los Capítulos XI al XIV.

El capítulo XI propone algunas posibilidades de integrar MAOO y SOOO como la codificación de la funcionalidad del SOOO dentro de la propia MAOO.

La conclusión de este capítulo supone la adopción de una arquitectura reflectiva que integre y combine ambos elementos para lograr un entorno de computación OO flexible.

Dada la trascendencia de esta arquitectura, como elemento aglutinador en el SIOO, los capítulos XII y XIII se dedican a presentar en profundidad, los conceptos teóricos referidos a la reflectividad. Desde los términos más fundamentales hasta las organizaciones reflectivas, así como las ventajas e inconvenientes de cada esquema.

Por último, el capítulo XIV aplica todos estos conceptos a la organización concreta de la MAOO y el SOOO presentados, decidiendo el número de niveles de la torre, la arquitectura reflectiva propuesta, el paso de control del nivel base al meta nivel y viceversa, y, en general, todas las decisiones a las cuestiones planteadas en el capítulo XII.

Así, la MAOO se convierte en el nivel más básico mientras el SOOO refleja o expone el comportamiento de ciertos aspectos de la máquina abstracta.

Este reflejo se hace siguiendo el modelo meta objetos, lo que supone que, dichos componentes de la MAOO se representan mediante objetos homogéneos con el resto de los objetos del sistema.

El paso de control del nivel base al meta nivel se produce ante la ocurrencia de determinados eventos, fundamentalmente, el paso de mensajes. La ejecución en el meta nivel está controlada por la MAOO, lo que supone la existencia de dos niveles en la torre reflectiva, suficiente para dotar al sistema de flexibilidad y no suponer una sobrecarga notable.

1.2.4 Máquina Abstracta y Sistema Operativo por dentro

Los últimos capítulos de esta tesis están dedicados a profundizar en la máquina abstracta y el Sistema Operativo propuestos.

Máquina Abstracta Orientada a Objetos

El capítulo XVI está dedicado, íntegramente, a describir la MAOO propuesta, desde el ensamblador o lenguaje máquina hasta la jerarquía de clases básica codificada a bajo nivel en la propia máquina y el funcionamiento interno de la misma.

Se describen más detalladamente aquellos aspectos de la máquina abstracta que más directamente influyen en la computación de objetos.

En primer lugar se describe la forma en que la máquina da soporte a la abstracción de objeto activo, mediante la definición de objetos internos, ligados a los objetos, que describen la ejecución de sus métodos.

En segundo lugar, se define el mecanismo de comunicación, no sólo como tal, sino como medio de introducir la concurrencia en el modelo mediante la definición de una semántica wait-by-necessity, y como medio para transferir el control al sistema operativo, es decir, como herramienta básica para la consecución de la reflectividad del comportamiento.

En tercer lugar, se describe la jerarquía de clases para la reflectividad del comportamiento. Gracias a estas clases, que se instanciarán por objetos en tiempo de ejecución, la máquina abstracta expone al exterior su propia maquinaria interna, permitiendo que el usuario no sólo la consulte, sino que en algunos casos, la manipule.

Esta jerarquía reflectiva, junto con la definición del paso de mensajes y otros eventos que provocan el paso de control a los objetos del sistema operativo, constituyen el núcleo de la máquina que da soporte a la reflectividad del comportamiento.

Sistema Operativo Orientado a Objetos

Por su parte, el Capítulo XVII está dedicado a describir el Sistema Operativo. Como conjunto de objetos que, gracias a la reflectividad extienden el comportamiento de la máquina abstracta, esta descripción se fundamenta en conocer qué objetos forman el Sistema Operativo, la interfaz de estos objetos, los medios para modificar el Sistema Operativo y la forman en que el control pasa del nivel base al sistema operativo y viceversa.

El Sistema Operativo está compuesto por cuatro objetos, Emisor, Receptor, Sincronizador o Ejecutor y Planificador. Cada uno de estos objetos describe una parte del comportamiento de la máquina abstracta.

Emisor y Receptor describen el comportamiento de la máquina ante el envío de mensajes, matizándolo al realizar antes y después del envío efectivo del mismo, diversas acciones destinadas a gestionar excepciones producidas durante el mismo, a decidir qué hacer en casos excepcionales o a determinar el mejor método a ejecutar como respuesta al mensaje recibido.

Sincronizador expone el modelo de objetos para la concurrencia determinando el nivel de concurrencia permitido en el objeto. En caso de que varios métodos puedan ejecutarse simultáneamente en el objeto base, el objeto Sincronizador determinará las normas para la correcta ejecución de los mismos, garantizando la consistencia del estado interno del objeto.

Por último, el planificador expone la política de planificación de la máquina abstracta. Esta política no tiene por qué ser única e inamovible. Se permite la creación de jerarquías de planificadores en tiempo de ejecución que se ceden el tiempo de la máquina y que implantan distintas políticas con el fin de adaptarse al máximo a las necesidades de las aplicaciones.

Todos estos objetos pueden necesitar, a su vez, definir otros objetos, desde colas de recursos hasta semáforos, etc. Al estar definidos en el lenguaje de la máquina, se permite que los usuarios puedan definir un nuevo objeto de alguna de las cuatro categorías mencionadas.

Este nuevo objeto debe mantener la interfaz por defecto pero puede implantar la funcionalidad como el usuario desee.

De esta forma se permite la personalización del entorno OO a las necesidades de aplicaciones específicas.

Prototipo

Por último, el Capítulo XVII describe uno de los posibles prototipos del entorno integral, máquina abstracta y sistema operativo.

Nótese nuevamente que no se ha dado importancia a la implantación eficiente del mismo, sino al hecho en sí de su implantación, mostrando así que la construcción de un sistema como el descrito es posible.

Conclusiones

Por último, los capítulos XVIII y XIX se dedica a recopilar las ideas más destacables que se derivan de esta tesis, así como a resaltar los caminos de investigación que se abren a partir de la conclusión de la misma.

2 Resultados más Destacables

Los resultados derivados de esta tesis son varios y en distintos apartados. Desde la construcción de un SIOO como medio de soporte básico a la abstracción de objeto, hasta la elección de un modelo de reflectividad adecuado a los requisitos impuestos.

Sin embargo, a continuación se hace referencia únicamente a los más destacables, aquellos que tienen una mayor trascendencia.

2.1 SIOO=MAOO+SOOO. Soporte Global al objeto

El primer resultado a destacar es la propia elección de un SIOO como plataforma de soporte a los objetos. Este sistema integral, compuesto de Máquina Abstracta y Sistema Operativo, se revela como solución global a los problemas de interoperabilidad de objetos presentados.

Soporte Integral al Objeto

El SIOO proporciona soporte a la abstracción de objeto en tiempo de ejecución. Es decir, aunque la aplicación de las tecnologías OO tanto en la fase de diseño como en la implantación de máquina abstracta y Sistema Operativo, producen una importante cantidad de ventajas, el objeto no es simplemente una herramienta de diseño o un elemento del lenguaje de programación, sino una entidad en tiempo de ejecución.

La infraestructura para este mundo de objetos proporciona mecanismos para la existencia del objeto, la identificación única y la interacción.

El Sistema Integral ofrece una representación interna del objeto que permite crear instancias de una clase dinámicamente, proporcionar nuevas descripciones o clases, eliminarlas, modificarlas, etc. También identifica unívocamente a los objetos mediante identificadores internos que permiten la interacción entre ellos por medio únicamente, del paso de mensajes.

Uniformidad

La definición del Sistema Operativo como un conjunto de objetos, no ya en tiempo de diseño o construcción, sino en tiempo de ejecución, hace que el principio de uniformidad se mantenga hasta sus últimas consecuencias.

La funcionalidad del Sistema Operativo estará ofrecida por un conjunto de objetos, uniformes con el resto de los objetos del sistema, ya sean estos de usuario, o de otras capas del SI.

La solicitud de servicios al Sistema Operativo, lo que tradicionalmente se conoce como llamadas al sistema, se traduce por invocaciones a métodos en estos objetos.

Extensibilidad del sistema

El sistema es fácilmente extensible, tanto estática como estáticamente.

- **Extensibilidad Estática**. La herencia y el polimorfismo permiten la definición de nuevos elementos de la jerarquía de clases que pueden sustituir a otros que no se adecuen a los requerimientos de las aplicaciones subyacentes.
- **Extensibilidad Dinámica**. La utilización de la reflectividad como organizador de MAOO y SOOO permite la modificación del comportamiento de la máquina en tiempo de ejecución.

2.2 Modelo de objetos Activo: Uniformidad total

La definición de un modelo de objetos activo, que transforma el objeto en una herramienta de computación autónoma, tiene dos lecturas diferentes.

Soporte más Sofisticado a la abstracción de Objeto

Por un lado, se necesita un soporte más sofisticado que el habitual a la abstracción de objeto. El objeto deja de ser un mero contenedor de datos y métodos para convertirse en un ejecutor de aquellos métodos que forman su interfaz y que, por tanto, actúan sobre su estado interno.

El modelo de objetos activo propone el soporte a una abstracción de objeto que amplía el concepto tradicional para añadir al mismo las actividades internas destinadas a la ejecución de sus métodos.

La propia máquina abstracta ofrece una visión de objeto activo, de tal forma que las actividades se transforman en objetos internos de la máquina en tiempo de ejecución que representan la ejecución de métodos en los objetos, y cuya existencia está ligada a ellos desde su inicio hasta su finalización.

Uniformidad Total en el Sistema

El modelo activo unifica en una única abstracción las entidades necesarias para la computación en el sistema, proporcionando a este una simplicidad máxima y una visión natural del mismo.

Por ello, el SIOO ofrece una única abstracción, el objeto, y no existe ninguna diferencia entre objetos de usuario o de sistema ni entre unos objetos que tienen capacidades de ejecución de las que otros carecen. Se logra la uniformidad total en el sistema.

Simplicidad y Economía de Conceptos

El problema del mundo real tiene un reflejo en el sistema como un conjunto de objetos que se comunican mediante el paso de mensajes, y que ejecutan métodos en respuesta a estos.

No existe una supra-entidad paralela al objeto, como podría ser el proceso, que refleja el flujo de control para la ejecución de métodos, con la que los programadores deben introducir la concurrencia en el modelo.

La simplicidad de conceptos definidos, determina que sea un modelo fácilmente comprensible y que no exija un esfuerzo importante a los programadores para comprenderlo y utilizarlo.

2.3 Integración de MAOO y SOOO por medio de la Reflectividad. Flexibilidad

El entorno integral se compone de una máquina abstracta y un sistema operativo.

La máquina abstracta ofrece la funcionalidad básica, aquella que resultaría caro, peligroso o difícil implantar en cualquier otra capa del Sistema Integral.

El Sistema Operativo extiende el nivel básico, es decir, añade o modifica la funcionalidad codificada internamente en la máquina, adaptando el comportamiento de esta al entorno concreto.

La necesaria integración de ambos niveles puede llevarse a cabo aplicando varios principios. Sin embargo, la **reflectividad** se ha mostrado como el mecanismo más

prometedor para obtener un entorno de computación flexible de la suma de una máquina abstracta y un sistema operativo.

La aplicación de la reflectividad como pegamento para unir MAOO y SOOO, provoca que este último, se convierta en la capa de extensión de la primera. Así, funcionalidad que, en principio se implanta en la MAOO se implanta ahora como objetos homogéneos con cualquier otro objeto del mundo de objetos.

Extensión Uniforme con el Resto del Sistema

El Sistema Operativo ofrece su funcionalidad mediante objetos no diferenciables de los objetos definidos por los usuarios y que se invocan de forma similar (**extensión uniforme con el resto del sistema**) ofreciendo un entorno similar al que ofrecen las aplicaciones de usuario.

Entorno Flexible

La configuración del Sistema Operativo como un conjunto de objetos normales en tiempo de ejecución, hace posible la modificación del entorno, al ser posible sustituir un objeto por otro, sin grandes dificultades.

La única exigencia es que el nuevo objeto responda a la misma interfaz, aunque puede implementarla de forma muy distinta al objeto original.

Igualmente, al ser el sistema operativo un conjunto de objetos instanciados a petición de los objetos de las aplicaciones, es posible modificar una característica del Sistema Operativo para un objeto o conjunto de objetos, sin afectar al comportamiento del sistema Operativo para el resto del sistema.

Esta posibilidad de modificación personalizada del sistema en tiempo de ejecución, dota al entorno de una flexibilidad de la que carecen otros entornos.

2.4 Máquina Abstracta: Base de la torre reflectiva

La estructura propuesta gira alrededor de una **máquina abstracta**, nivel básico que ofrece la funcionalidad mínima necesaria para el soporte a la existencia de los objetos, la ejecución de sus métodos y su comunicación.

En este aspecto, se asemeja a un exokernel, minimizando la funcionalidad implementada en la máquina para hacer posible la flexibilidad.

La máquina abstracta implementa el papel de meta espacio básico, al finalizar la reflectividad.

2.5 Sistema Operativo: Meta Nivel

El **sistema operativo** extiende la funcionalidad básica ofrecida por la máquina abstracta, por medio de un conjunto de objetos hasta proporcionar un entorno de computación adecuado.

El Sistema Operativo se convierte en el meta-nivel, reflejo de algunos aspectos de la máquina abstracta mediante un conjunto de objetos y especifica que cada objeto del nivel base o nivel de la aplicación tiene asociado un meta-nivel conceptual.

Esta asociación es dinámica y puede eliminarse o cambiarse en cualquier momento en tiempo de ejecución.

Permite modificar la semántica del objeto dinámicamente, lo que confiere gran flexibilidad al entorno.

Aspectos de la máquina expuestos por el Sistema Operativo

El sistema operativo propuesto expone, fundamentalmente, aquellos aspectos de la máquina que se refieren a la computación de objetos.

- Paso de Mensajes. Se puede personalizar tanto las acciones que se llevan a cabo para el envío como para la recepción de mensajes, adaptándolas al entorno concreto.
- Procesamiento del Mensaje. La ejecución de métodos también se expone al meta nivel que decide qué método ejecutar como respuesta a un mensaje determinado, si ejecutarlo inmediatamente o retrasarlo, etc.
- Planificación de Actividades.

Capítulo XIX

Futuras Líneas de Trabajo más Prometedoras

La integración de las TOO y los Sistemas Operativos es un área muy joven todavía. Si a ello sumamos la novedosa propuesta de construir un SIOO basándose en un SOOO que se apoya en una MAOO, que ofrece un modelo de objetos activo, se comprenderá que sean muchos los frentes que se dejan abiertos en esta investigación.

Si a ello sumamos la utilización de la reflectividad como elemento de unión entre la máquina abstracta y el Sistema Operativo, las líneas futuras de trabajo aumentan.

Por tanto, son varias las líneas por donde se podría continuar este trabajo. Desde la ampliación de los aspectos expuestos de la máquina abstracta, el diseño de lenguajes de alto nivel que aprovechasen las características reflectivas que se ofrecen, la construcción de sistemas de agentes sobre el modelo de computación basado en objetos activos que permite migrar la ejecución de un objeto y otras.

De todas ellas se comentan aquí las más prometedoras.

1 Reflectividad Bajo Demanda

Una de las premisas a la hora de diseñar y construir un sistema reflectivo es que la reflectividad no debería provocar ninguna sobrecarga si no se usa el meta nivel. Es decir, en el sistema reflectivo diseñado, debe pagarse, únicamente, por aquella reflectividad que se utiliza.

Esto supone, que el meta nivel sólo debe utilizarse si el objeto del nivel base considera necesario personalizar ese aspecto de su entorno.

En caso contrario, debería ser la máquina la que, funcionando como meta objeto básico, llevase a cabo las acciones necesarias.

Dado que el objetivo era comprobar la posibilidad de construir un sistema reflectivo, el trabajo actual se centró en la posibilidad de extraer características de la máquina y representarlas como objetos de alto nivel, dejando a un lado la posibilidad de mantener dichas características implantadas a bajo nivel en la máquina, si el nivel base así lo deseaba.

Por tanto, uno de los trabajos futuros más interesantes es implantar distintos prototipos en los que las máquinas abstractas subyacentes implanten un Protocolo básico distinto y comparar su rendimiento.

2 Diseñar y Construir compiladores de lenguajes de programación OO

Obviamente, los usuarios del SIOO no construirán sus aplicaciones utilizando directamente el lenguaje ensamblador de la máquina abstracta, sino utilizando para ello Lenguajes de Programación de alto nivel OO.

Partiendo de este hecho, uno de los futuros trabajos consiste en diseñar y construir compiladores para tales lenguajes que saquen partida del entorno reflectivo subyacente.

La investigación en este campo se puede ver desde dos puntos de vista distintos: la adecuación del sistema reflectivo y el diseño del propio lenguaje en aquellos aspectos que impliquen el uso de características reflectivas.

2.1 Adecuación del Sistema Reflectivo subyacente

En primer lugar, la construcción de capas de más alto nivel que trabajen sobre el sistema reflectivo (máquina abstracta más sistema operativo) propuesto, puede sacar a la luz deficiencias del sistema que necesitarán ser subsanadas para su uso intensivo.

2.2 Análisis de las categorías reflectivas

En segundo lugar, el diseño del lenguaje, desde el punto de vista de las características reflectivas que se añadan al mismo, puede proporcionar información para posteriores implantaciones más eficientes.

Partiendo de la premisa de que no es obligatorio exponer todas las características, sino que el usuario debe manifestar su interés en alguna de ellas, en todas o en ninguna, se crean Categorías Reflectivas para indicar al compilador en qué aspecto se debe producir la reflexión del sistema subyacente.

La creación de las Categorías reflectivas y su uso intensivo pueden proporcionar valiosa información acerca del sistema reflectivo subyacente. Esta información puede, no sólo sacar a la luz alguna carencia del sistema integral, sino líneas de diseño o implantación más adecuadas que las actuales con el fin de lograr un diseño más eficiente.

3 Construcción de una jerarquía del Sistema Operativo más completa

Aunque se ha construido un Sistema Operativo, este es muy básico. Cuenta con una clase para cada aspecto reflejado, paso de mensajes, ejecución de métodos y planificación.

Sería conveniente ampliar la jerarquía de clases del sistema operativo ofreciendo distintos modelos de paso de mensajes, integrándolo con la persistencia y la distribución, para lograr la tan ansiada transparencia de localización de objetos.

De la misma forma, la jerarquía de planificadores esta formada por un planificador FIFO y un planificador RR que se comporta como raíz de la jerarquía. La extensión de esta jerarquía es también un trabajo abierto.

Por último, la clase Sincronizador limita el comportamiento del objeto a la ejecución de un método simultáneamente. La adición de nuevas clases permitirían introducir un nivel de concurrencia mayor con la considerable mejora en el rendimiento del sistema en tiempo de ejecución.

4 Reflectividad guiada por eventos

Un trabajo muy interesante sería la modificación de la MAOO subyacente para integrar en ella un modelo de eventos semejante al de la máquina Java 1.2???

➔ es esa la versión??

A partir de esta nueva máquina abstracta se podría modificar la implantación de la reflectividad del comportamiento haciendo que esta se produjese en función de los eventos de la máquina.

Para ello, el usuario que desease reflejar algún aspecto, debería suscribirse a determinado evento de la máquina, que provocaría el paso de control al meta nivel.

De hecho, el paso de control sería modificable dinámicamente, pues sería el propio usuario el que diría cuántos y cuáles eventos provocarían la ejecución de un determinado método de un meta objeto del Sistema Operativo.

5 Integración de otros aspectos en la arquitectura reflectiva propuesta

Algunas de las premisas que más relevancia tienen a la hora de diseñar un sistema reflectivo son, en primer lugar, que no debe suponer una sobrecarga excesiva para el sistema, lo que implica que el meta nivel de debe llevar a cabo cosas importantes. Por otro lado, la arquitectura reflectiva propuesta debe ser lo suficientemente general para integrar varios aspectos.

Aunque ya se ha comprobado que, con la arquitectura definida, es factible integrar aspectos diversos, como la comunicación entre objetos, la planificación de actividades o el modelo de objetos para la concurrencia, por ejemplo, sería muy interesante integrar en ella otros aspectos importantes en un sistema integral orientado a objetos y que, actualmente, se están llevando a cabo de forma independiente.

Estos aspectos son, fundamentalmente, aspectos de la distribución, la Persistencia y la seguridad del sistema.

Distribución

Se pueden implantar diversos aspectos de la distribución en el meta nivel. Desde la política de actuación cuando un objeto requerido no se encuentra hasta la migración de objetos como respuesta a alguna falta de objeto.

Por ejemplo, el meta nivel se puede emplear para controlar la estrategia de migración de objetos. Así, el modelo propuesto en [Tok90], se puede implementar en el meta nivel, tal y como se muestra en [OI94].

Persistencia

De la misma forma, la política de persistencia/recuperación aplicada a los objetos, que no tiene por qué ser la misma para todos, también se puede implantar en el meta nivel.

Así, por ejemplo, ante una falta de objeto, puede programarse en el meta nivel la búsqueda local o remota del mismo.

Agrupación de objetos

Otra posibilidad, íntimamente relacionada con las dos anteriores, distribución y persistencia, es la implantación de grupos o agrupaciones de objetos en el meta-nivel.

La agrupación de objetos puede, en ocasiones, proporcionar beneficios importantes. Así, por ejemplo, un meta-objeto puede gestionar la migración de varios objetos de forma conjunta o el movimiento de los mismos a almacenamiento secundario.

Gracias a la separación de nivel base y meta nivel, esta agrupación artificial, se lleva a cabo de forma totalmente transparente al nivel base. Sólo el meta nivel es consciente de ella.

Seguridad

También se pueden utilizar los meta-objetos para implantar varias políticas de seguridad.

Por ejemplo, se puede asociar un meta objeto a un objeto antes de pasarlo a un objeto inseguro, para controlar el acceso y propagación del mismo. Ese meta objeto puede utilizarse para expirar o revocar derechos de acceso al objeto.

Los meta objetos también se pueden utilizar como guardianes para comprobar todos los accesos al objeto específico.

6 Implantación eficiente del prototipo

Ya se mencionó en distintos puntos en este mismo documento, que la eficiencia no era un aspecto que gozase de mucha importancia entre nuestros requisitos.

Sin embargo, es un campo de trabajo con muchas posibilidades.

Anexo A

Descripción del lenguaje Carbayón: Juego de Instrucciones

En este anexo se presenta el juego de instrucciones proporcionado por la máquina abstracta Carbayonia, como descripción del lenguaje ensamblador asociado al mismo.

Este lenguaje de bajo nivel recibe el nombre de **Carbayón** y constituye la interfaz que la máquina ofrece a las aplicaciones para la utilización de sus recursos con vistas a su ejecución.

Para la descripción del lenguaje se utiliza un convenio de representación que facilita la completa descripción del mismo de forma breve. Entre las reglas que forman este convenio se encuentran las siguientes:

- El código del lenguaje se representa con un tipo de letra especial.
- Las palabras en negrita denotan **palabras reservadas**.
- Las palabras entre símbolos < y > denotan una <cadena de texto>, como el nombre de una clase.
- Las llaves {} denotan repetición del elemento que encierran (una lista separada por comas o por punto y coma).
- Los corchetes [] denotan opcionalidad del elemento que encierran.
- Pueden introducirse comentarios mediante el carácter "/". Todos los caracteres que se encuentren desde ese carácter hasta el fin de línea se ignorarán.

1 Instrucciones declarativas: Descripción de las clases

La clase es el elemento básico proporcionado por la máquina para definir la funcionalidad de una aplicación. Por ello, las primeras instrucciones que se describirán , son aquellas que se refieren a la descripción de las clases.

Para describir completamente una clase es necesario definir su nombre, las relaciones que guarda con otras clases, ya sean estas relaciones de herencia, agregación o genéricas, y los métodos que la componen.

La descripción en Carbayonia es la siguiente:

```
Class <Nombre>
Isa {Clase}
Aggregation {Nombre: Clase;}
Association {Nombre: Clase;}
Methods
  {<Nombre> ([{Clase}])[:Clase] <cuerpo>}
EndClass
```

A continuación se describen más detalladamente todos los aspectos de la declaración de una clase.

1.1 Identificación de una Clase

La instrucción declarativa **Class**, con la que comienza cualquier definición de una clase, sirve para identificar la clase a través del **nombre** que la acompaña. Este nombre deberá ser único y distinto a todos los nombres de las clases previamente almacenadas en el área de Clases.

1.2 Relaciones entre Clases

1.2.1 Relaciones de Herencia

Para especificar las relaciones de herencia que la clase a definir guarda con otras clases, se introduce la instrucción **isa** nombre_clase.

Aquí se indican todas aquellas clases de las que hereda la clase que se está definiendo. Es posible indicar más de una clase, lo que indica que se permite la **herencia múltiple**.

Duplicidad de métodos y variables

Se trata ahora el caso de que una clase A herede de dos clase B y C, definiendo ambas un método m. Si una instancia de la clase A invoca a dicho método, se ejecutará el método de la clase que primero aparezca en la sección **isa** de la clase A. En caso de que se desee acceder al otro método se deberá especificar antes del nombre del método la clase a la que pertenece.

```
a.M()       // método de la clase B
a.B:M()     // método de la clase B
a.C:M()     // método de la clase C
```

1.2.2 Relaciones de Agregación

Las relaciones de agregación definen qué objetos componen la clase, que serán aquellos que aparezcan a continuación de la palabra reservada **aggregation**. Realmente, se indican referencias a los objetos ya que, en Carbayonia, todo trabajo con un objeto se hace a través de una referencia al mismo.

Para cada uno se indica el nombre con el que se les va a hacer referencia a lo largo de la definición de la clase y la clase a la que pertenece. Estos objetos se crean y destruyen automáticamente cuando se crea y destruye el objeto que los contiene, siendo este además el único modo de eliminarlos.

1.2.3 Relaciones de Asociación

Por último, en la sección de definición de relaciones, se utiliza la palabra clave **Association** para indicar las relaciones de asociación que existen entre la clase a definir y otras.

Se indican así los nombre de las referencias que juegan este rol en la clase que se está definiendo.

A la hora de definir una clase, es necesario recordar que las referencias asociadas, a diferencia de las agregadas, son responsabilidad del programador. Es decir, mientras que los agregados son creados y destruidos automáticamente, los agregados deben crearse y destruirse explícitamente.

1.3 Métodos

Por último, en la definición de cualquier clase, es necesario definir los métodos que componen su interfaz, indicando su cabecera y su cuerpo.

La instrucción **methods**, sirve para declarar los métodos que componen la clase, indicando la cabecera de los mismos, aunque la definición se haga más tarde.

La definición es bastante estándar. Se trata de dar un nombre al método y, a continuación y entre paréntesis, los parámetros, identificado por la clase a la que pertenece. Si existe valor de retorno, al final se indica la clase del mismo.

Finalmente, se incluye un pequeño ejemplo de declaración de clases en Carbayonia, con el fin de que el lector pueda tener una idea más precisa.

Siendo el siguiente diagrama representativo de la clase y las relaciones que se desean representar:

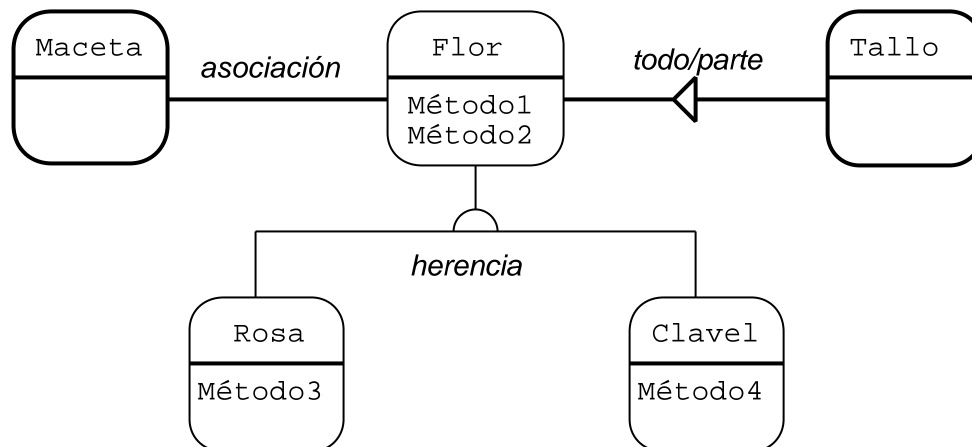


Figura A1.1 Diagrama con los tres tipos de relaciones.

La codificación del mismo en Carbayonia sería:

```
/---------------------
class Flor
isa object  /La razón se verá posteriormente
aggregation
      t: Tallo;
association
      m: Maceta;
```

```
methods
      Metodo1(clase1);
      Metodo2(clase2, clase3): clase4;
endclass

/---------------------
class Rosa
isa Flor
methods
      Metodo1(clase1);  // Redefinición de Metodo1
      Metodo3(clase4);
endclass

/---------------------
class Clavel
isa Flor
methods
      Metodo2(clase2, clase3): clase4;    /Redef. de Metodo2
      Metodo4(clase5);
endclass
```

2 Definición de métodos

Una vez superada la declaración de la clase, es necesario definir el comportamiento de cada uno de los métodos que la componen. La definición de un método consiste en una parte declarativa, indicando el nombre, argumentos y referencias locales, y un cuerpo, en el que se indican las instrucciones que constituyen el método.

2.1 Cabecera de método

La forma de describir la cabecera de un método es como sigue:

```
<Nombre> ([{Clase}])[:Clase]
[Refs {<Nombre>:Clase};]
[Instances {<Nombre>:Clase};]
```

2.1.1 Identificación del método

Al igual que en la definición de la clase, la definición del método comienza dando el nombre del mismo y los argumentos, siendo estos referencias para las que se indican nombre y clase que debe encajar con la que se define en la sección de definición de métodos de la clase.

2.1.2 Referencias

Con la palabra reservada **Refs** comienza la definición de las referencias locales necesarias para la adecuada definición del método, indicando nombre y clase. Estas referencias locales se crean automáticamente al entrar a ejecutar el método, y se destruyen automáticamente cuando se alcanza la última instrucción del mismo y su ámbito se reduce al método en que se definen.

Aunque se deja en manos de la máquina abstracta su creación y destrucción automática, no así su inicialización. Es decir, es necesario determinar el objeto al que referencian e indicarlo explícitamente en el cuerpo del método.

```
Refs
  i: Integer;
  b: ClaseB;
```

2.1.3 Instancias

La instrucción **instances** señala el punto en que se declaran algunas instancias que se utilizan en la construcción del método, aquellas cuya gestión se deja a la máquina abstracta.

Nótese que guarda mucha similitud con la sección anterior, salvo que en este caso, la máquina abstracta no sólo gestiona las referencias, sino además crea y destruye automáticamente al entrar y salir del método, la instancia a la que apunta la referencia.

Al igual que en el caso anterior, tanto la referencia como el objeto tienen ámbito local al método en que se definen.

```
Instances
  i: Integer;
  b: ClaseB;
```

Diferencia Refs, Instances

Nótese que, excepto por la palabra reservada instances, no hay ninguna diferencia con la sección de referencias. Sin embargo, existen diferencias sustanciales en la implantación de ambas instrucciones.

La instrucción instances no sólo inicializa las referencias y los objetos, sino que también se ocupa de liberar ambos. Sin embargo, la instrucción Refs sólo libera las referencias, y no se ocupa de eliminar los objetos a los que apunten dichas referencias, hecho este que queda, totalmente, en manos del programador.

No existe, por tanto, una gestión automática de memoria (recolector de basura) que se ocupe de realizar esta tarea.

2.2 Cuerpo del método

Tras la descripción del nombre, parámetros, referencias e instancias locales que va a usar el método, llega el turno de definir el cuerpo del método propiamente dicho, lo que se lleva a cabo indicando ordenadamente las instrucciones que lo componen. El cuerpo del método viene delimitado por las palabras reservada **Code** y **EndCode**.

La forma de describir el cuerpo de un método es como sigue:

```
Code
  <Codigo>
EndCode
```

Al igual que en muchos lenguajes, se sigue la norma de que todas las instrucciones terminan con un punto y coma.

3 Juego de Instrucciones de la máquina

La máquina abstracta Carbayonia se define como una máquina tipo RISC, con un juego de instrucciones reducido.

A continuación se describen las instrucciones que componen la máquina pero, en lugar de ofrecer una enumeración detallada de las mismas, se ha preferido mostrarlas agrupándolas por un criterio de funcionalidad.

3.1 Instrucciones Referidas a la Manipulación de Objetos

Una de las partes más importantes del juego de instrucciones lo constituyen aquellas instrucciones que permiten crear, destruir y manipular objetos. Dado que todo trabajo con un objeto debe realizarse siempre a través de una referencia, se incluyen aquí aquellas instrucciones de manipulación de referencias, que permiten crear y liberar referencias o enlazar objetos y referencias.

3.1.1 Creación y Destrucción de Objetos

Creación de objetos

La creación de objetos, mediante la instrucción **new** <referencia>, además de crear un nuevo objeto de la clase que indique la referencia, enlaza el objeto recién creado con la referencia que se pasa como parámetro a new.

```
New <Referencia>

New <refString>, <Referencia>
```

En caso de que no se pueda crear el objeto, se lanza una excepción (por ejemplo por una falta de espacio).

Recuérdese que, al crear un objeto, la máquina abstracta se encarga de crear, automáticamente, todos los objetos especificados en la instrucción aggregation y los que se declaren en la sección **Instances**.

Por tanto, el campo de trabajo natural para la instrucción new son todas aquellas referencias declaradas que, inicialmente, no hayan sido ligadas a ningún objeto, es decir, aquellas declaradas en association y en refs.

Una característica muy interesante de la instrucción new es que permite determinar el tipo del objeto a crear **en tiempo de ejecución**. Para ello, la instrucción new admite una segunda forma: new <cadenaCaracteres>, <Referencia>, en la que, como primer parámetro se indica, en forma de una cadena de caracteres, el nombre de la clase de la que se creará el objeto. Será en tiempo de ejecución cuando se resolverá dicho nombre, transformándolo en una clase determinada indicada por el string.

Eliminación de objetos

Para eliminar una instancia, se introduce la instrucción **Delete**:

```
Delete <Referencia>
```

Delete elimina el objeto al que apunta la referencia y producirá una excepción en los siguientes casos:

- La referencia está libre.
- La referencia apuntaba a un objeto que ya ha sido eliminado.
- Se intenta liberar un objeto agregado. En apartados anteriores se indicó que, tanto la creación como la destrucción de estos objetos lo lleva a cabo automáticamente la máquina abstracta, cuando se crea o libera el objeto al que pertenece.

3.1.2 Asignación de objetos a referencias

Instrucción new

Como se ha visto, una forma de asignar objetos a referencias es mediante la instrucción new.

Instrucción assign

Sin embargo, hay casos en los que se dispone de una referencia libre, no enlazada todavía a ningún objeto.

Otra posibilidad de enlazar una referencia y una instancia es asociarlos explícitamente mediante una instrucción que apunte la referencia al objeto deseado.

Evidentemente, dado que todo trabajo con objetos se realiza mediante referencias, la asociación entre la referencia libre y el objeto debe realizarse mediante una segunda referencia que se encuentre apuntando a la instancia.

La instrucción **Assign** se encarga de llevar a cabo estas asignaciones explícitas.

```
Assign <ReferenciaDestino>, <ReferenciaFuente>
```

Al finalizar la instrucción el objeto se podrá manipular por cualquiera de las dos referencias indistintamente. Se producirá una excepción si la referencia destino no es compatible con el tipo del objeto al que apunta la referencia fuente.

Asignaciones implícitas en los parámetros de los métodos

Una tercera opción para asignar objetos a referencias se basa en la asignación implícita que la máquina abstracta realiza con los parámetros que se pasan a los métodos.

Valor de retorno de un método

Otra posibilidad de asignar objetos a referencias es la que proporciona el retorno de un método. En este caso es posible recoger en una referencia el valor de retorno del mismo.

Esta asignación se lleva a cabo de la forma siguiente:

- **Finalización del Método**. Para devolver el objeto resultado de un método se utiliza la referencia **rr** (**referencia de retorno**), presente en cualquier contexto de ejecución. Para ello, en el cuerpo del método se asigna a la referencia rr el valor de retorno que se desee.

```
Assign rr, ObjetoRetorno
```

```
Exit
```

- **Asignación Implícita**. La máquina asigna la referencia rr al parámetro que se utiliza para recoger el valor de retorno.

Referencia del Sistema this

Por último, existe una referencia especial, **this**, que forma parte del conjunto de **referencias del sistema** y que la máquina abstracta se encarga de manipular de forma automática asignándole siempre el objeto actual.

Gracias a esto, se puede realizar operaciones sobre el objeto actual (asignar a otras referencias, invocar métodos, etc.) igual que se haría con otros objetos[1].

```
this.Metodo(parámetro(s))/ Invoca método en el objeto actual
```

Comprobación de tipos en tiempo de ejecución

Carbayonia realiza **comprobación de tipos en tiempo de ejecución**, lo que resulta bastante más potente y flexible que la comprobación usual en tiempo de compilación.

Esto permite aplicar, durante la ejecución, una serie de reglas implícitas de conversión de tipos en la asignación de referencias, codificadas en la máquina abstracta, al estilo de las normas que tienen otros lenguajes para el mismo fin, como C o C++.

Al igual que en estos lenguajes, en una asignación, el tipo del operando a la derecha del operador de asignación, se convierte al tipo del operando a la izquierda.

Así, si tenemos una referencia r_1 de tipo T_1 y una referencia r_2 de tipo T_2, donde T_2 isa T_1, se permite:

- **Promoción**. Se permite la asignación del tipo assign r_1, r_2. Este es el caso habitual en el que la clase derivada se asigna a la clase base.
- **Degradación**. Sin embargo, Carbayonia también permite la instrucción assign r_2, r_1. Es precisamente en este caso en el que se aplica toda la potencia de comprobación de tipos **en tiempo de ejecución**. El intento de asignar una referencia de una clase base a una referencia de una clase derivada provoca la comprobación del tipo de la referencia r_1. Si r_1 es realmente una instancia de la clase T_2, en cuyo caso la ejecución de la instrucción assign no dará ningún problema, o no, en cuyo caso, se elevará una excepción.

3.2 Instrucciones Relativas a la Invocación de Métodos

Además de todo el juego posible con las referencias, la única operación posible sobre un objeto es la invocación de alguno de sus métodos.

[1] Por tanto, si se requiere acceder a sus variables miembro, no podrá hacerse directamente, lo que, claramente, rompería el principio de encapsulación, sino que deberán crearse métodos al efecto e invocarlos.

La ejecución de un método, de las instrucciones que componen el cuerpo de un método, consiste en la creación de nuevas instancias, la asignación de instancias a referencias y, fundamentalmente, en la invocación de métodos en dichas instancias.

La invocación de un método debe indicar la **referencia al objeto** del que se desea invocar un método, el **nombre del método** a invocar, los nombres de las referencias que actuarán como **parámetros** y, por último, si el método devuelve algún valor y se desea recogerlo para su posterior utilización, debe indicarse el nombre de una referencia a la que se asignará el **resultado de la invocación**.

No es estrictamente necesario que los parámetros utilizados en la llamada y retornados por la misma, coincidan exactamente con los tipos de los parámetros utilizados en la definición del método. Como se comentó anteriormente, la máquina abstracta aplica reglas implícitas de conversión, intentando realizar las asignaciones que se indican en el orden de los parámetros. Si es posible, realiza la asignación, si no, eleva una excepción.

```
<Referencia>.[<Ambito2>:]<Metodo>({<Referencia>})[:Referencia]
```

De manera equivalente, la siguiente también es una representación válida de la instrucción de invocación a métodos, utilizando en este caso un nombre explícito (**Call/Send**), para la misma[3].

```
Call <refObjeto> <refParam1> <refParam2> ... <refRetorno>

Send <refObjeto> <refParam1> <refParam2> ... <refRetorno>
```

3.3 Control de flujo

Las instrucciones de control de flujo se refieren a cómo evoluciona el flujo de procesamiento en la máquina. Por defecto, este evoluciona de forma secuencial, es decir, si no se indica lo contrario, la siguiente instrucción a ejecutar en el método actual será siempre la que se encuentra inmediatamente a continuación de la instrucción actual[4].

3.3.1 Finalización de un método

La primera y más sencilla instrucción de control de flujo es aquella que termina la ejecución del método actual lo que, como ya se ha mencionado anteriormente en este mismo anexo, provocará la liberación automática de las referencias y de las referencias e instancias de las secciones Refs e Instances, respectivamente.

```
Exit
```

[2] El ámbito de la referencia es opcional y su función principal es resolver las ambigüedades en caso de herencia múltiple, como ya se señaló anteriormente.

[3] Se prefiere esta segunda notación, por tanto ya se ha hecho referencia muchas veces a la instrucción Call a lo largo de esta tesis, por lo que se facilita la lectura del resto del documento.

[4] Se entiende aquí que se refiere a la ejecución de un método. Evidentemente, en un entorno concurrente en el que se permita el requisamiento, habrá que tener en cuenta las posibles interrupciones en la ejecución causadas por la planificación.

3.3.2 Salto incondicional

Las instrucciones de salto dirigen la ejecución del programa a otro punto dentro del mismo método.

El salto incondicional, ruptura de control sin condiciones, provoca que la ejecución continúe en una instrucción determinada por el parámetro etiqueta.

```
Jmp Etiqueta
...
Etiqueta: / Otras instrucciones
```

3.3.3 Salto condicional

Los saltos condicionales, al igual que los anteriores, rompen el flujo normal de ejecución de instrucciones, provocando que este continúe en la instrucción determinada con la etiqueta. Sin embargo, a diferencia del salto incondicional, se lleva a cabo un paso previo de comprobación de una condición, que determinará el salto o no. La forma general de estas instrucciones es:

```
CódigoInstrucción Condición, Etiqueta
```

Y la forma concreta de las mismas es:

```
JT[D] <RefBool>, Etiqueta
JF[D] <RefBool>, Etiqueta

JNull <Referencia>, Etiqueta
JNNull <Referencia>, Etiqueta

Etiqueta: / Otras instrucciones
```

Las dos primeras codifican la condición como el valor de un objeto booleano, que tomará el valor verdadero (True) o falso (False) y cuya referencia se pasa como parámetro.

Dependiendo del valor del objeto booleano, estas dos instrucciones deciden si saltan o no a la instrucción indicada por Etiqueta. JT saltará si el valor de la instancia es verdadero, mientras que JF lo hará si la instancia almacena valor Falso. El sufijo opcional **D** permite indicar, en caso de que se ponga, que el objeto bool se libera automáticamente por la misma instrucción de salto (pero no la referencia).

Las otras dos instrucciones codifican la condición como la inspección de si existe un valor asignado a una referencia, es decir, si la referencia está libre o no.

JNULL salta a la etiqueta si la referencia indicada como parámetro esta libre. JNNULL salta si la referencia no está libre.

3.3.4 Excepciones

Se definen dos instrucciones para el control de las excepciones: Handler y Throw.

```
Handler Etiqueta
```

```
...
Throw
```

La instrucción Handler indica dónde debe continuar el flujo de ejecución en caso de que ocurra una excepción, que será aquella que corresponda a la posición de la etiqueta en el código fuente.

Throw lanza una excepción. Cuando se ejecute un Throw, la ejecución del programa continuará en la dirección indicada por la última instrucción Handler ejecutada.

El lanzamiento de una excepción puede ser por un posible error en tiempo de ejecución (salirse del rango de un array) o bien porque el usuario la haya lanzado con la instrucción Throw.

Anexo B

Jerarquía de Clases Primitivas de la máquina

En este anexo se presenta la jerarquía básica de clases ofrecida por la máquina y que el programador podrá ir ampliando posteriormente. Todas estas clases, tanto la jerarquía básica ofrecida por la máquina, como aquellas definidas por el programador, se encuentran registradas en el área de clases, de forma permanente.

Para su manipulación se utilizan las instrucciones definidas en el Anexo A.

1 Clases Primitivas Básicas

1.1 Clase Object

La clase object es la raíz de la jerarquía y, por tanto, es la clase base de cualquier otra clase de Carbayonia. Todas las demás clases, ya sean primitivas o definidas por el usuario, derivan de ella aunque no se especifique implícitamente mediante isa.

Por tanto, toda clase de Carbayonia será un object y tendrá, entre otros, los métodos definidos por la clase object, redefinidos automáticamente por la propia Carbayonia cada vez que se declara una nueva clase.

La **interfaz de la clase Object** es la siguiente:

```
class object
methods

       getClass(): string;
       getID():integer;
       isa(string):bool;

endclass
```

El método **getClass** devuelve un string (clase que se verá posteriormente) que contiene el nombre de la clase a la que pertenece el objeto.

El método **getID** devuelve una instancia de un entero que identifica de forma única al objeto. De esta forma se puede averiguar si se tiene dos referencias apuntando al mismo objeto (aunque estas sean de distinto tipo).

El tercer método **–isa–** devuelve una instancia de un objeto de tipo bool que indica si la instancia pertenece a esa clase o a una derivada de ella.

Las principales funciones de esta clase son:

- permitir la construcción de contenedores genéricos. Al ser toda clase derivada de object (directa o indirectamente) toda instancia será un object, por lo que dicho tipo será el que se utilice en las estructuras de datos genéricas.
- realiza parte de las funciones propias del RTTI.

1.2 Bool

La clase bool representa al tipo booleano presente también en otros lenguajes. Los únicos valores válidos para esta clase son True y False y su importancia radica en que las instrucciones de salto utilizan instancias de la misma para decidir la dirección del salto (Ver Anexo A).

La interfaz de esta clase es:

```
class bool
isa object
methods

        setTrue();      / Pasa a valer TRUE
        setFalse();     / Pasa a valer FALSE
        Not();          / Intercambia su valor
        And(bool);      / AND entre su valor y el del parámetro
        Or(bool); / OR entre su valor y el del parámetro
        Xor(bool);      / XOR entre su valor y el del parámetro

endclass
```

Los métodos **setTrue** y **setFalse** carecen de argumentos y su función es establecer el estado de la instancia a True o False, respectivamente.

Not invierte el valor del objeto, de manera que si vale True pasa a valer False y viceversa.

El método **And** realiza un AND lógico entre el estado del objeto al que se le envía el mensaje y el estado del objeto que se le envía como parámetro (que deberá ser de tipo bool o derivado). De la misma manera los métodos **Or** y **Xor** realizan respectivamente el OR y el XOR (Or-exclusivo) entre el estado de la instancia que recibe el mensaje y el del parámetro.

1.3 Integer

La clase integer representa a un entero con signo y su interfaz es la que sigue.

```
Class integer
Isa object
Methods

   Add (integer);
   Sub (integer);
   Mul (integer);
   Div (integer);
   Set (integer);
   SetF (float);
   Equal (integer): bool;
   Greater (integer): bool;
   Less (integer): bool;

endclass
```

Siguiendo con la premisa básica de que el trabajo con objetos se realiza mediante la invocación de sus métodos, la máquina abstracta no define en el lenguaje instrucciones para operar con enteros, +, -, *, etc., sino que opta por definir en la clase básica integer, métodos cuya invocación provoca la modificación del valor almacenado en la instancia, add, sub, mul, etc.

Add, **sub**, **mul** y **div** se corresponden con las cuatro operaciones básicas. Realizan respectivamente la suma, resta, multiplicación y división del valor del objeto que recibe el mensaje por el valor del objeto que se le envía como parámetro. **Div** eleva una excepción si el parámetro vale 0.

Set y **SetF** implementan el operador de asignación y permiten asignar un valor al entero, bien sea a partir de otro entero o bien mediante la conversión de un float.

Por último, **equal**, **greater** y **less**, implementan las funciones de comparación de enteros. Reciben un parámetro de tipo entero y devuelven una instancia de tipo bool que indica el resultado de la comparación.

A continuación se muestra un ejemplo que sirve además para mostrar la forma de usar el método getID declarado en la clase object, cuya utilidad se basa en la comparación de igualdad con otro, obtenido de la misma manera. Dado que el identificador es único, si dos instancias devuelven el mismo identificador, debe tratarse de la misma instancia. Si no es así, simplemente se está tratando con dos objetos diferentes, pero no se puede deducir ninguna otra información de su valor.

<table>
<tr><th>Código en C</th><th>Código en Carbayonia</th></tr>
<tr><td><pre>Void prueba(claseX *param)
{
       if (this != param)
       {

       }
}</pre></td><td><pre>prueba(param:claseX)
refs
       id1, id2:integer;
       b:bool;
code
       this.getID():id1;
       param.getID():id2;
       id1.equal(id2):b;
       jtd b, iguales
distintos:
       // llegados a este punto
       // es que son distintos

iguales:
       exit
EndMethod</pre></td></tr>
</table>

1.4 Float

La clase float es idéntica a la clase integer excepto en que su estado es de tipo real en vez de un entero.

```
Class float
Isa object
Methods

   Add(float);
   Sub(float);
   Mul(float);
   Div(float);
   SetI(integer);
   Set(float);
   Equal (float): bool;
   Greater (float): bool;
   Less (float): bool;

EndClass
```

Los métodos **Add**, **Sub**, **Mul** y **Div** realizan respectivamente la suma, resta, multiplicación y división del estado del objeto por el estado del parámetro.

Los métodos **Set** y **SetI** sirven para establecer el estado del objeto, bien a partir de otro float, bien a partir de un integer, en cuyo caso se realizará una conversión de tipo.

Por último, los métodos **Equal**, **Greater** y **Less**, idénticos a los de la clase integer, establecen comparativas entre el valor del objeto y el valor de otro objeto pasado como parámetro devolviendo el resultado en una instancia bool.

1.5 String

La clase string representa a las cadenas de texto y su interfaz es:

```
Class string
Isa object
Methods

   Set(string);
   Length(): integer;
   Get (inicio, longitud:integer): string;
   Insert (string, integer);
   Delete (inicio, longitud:integer);
   Append (string);
   GetChar(integer):integer;
   setChar(integer, integer);
   toUpper();
   toLower();
   Equal (string): bool;
   Greater (string): bool;
   Less (string): bool;

Endclass
```

Set asigna el estado del parámetro al objeto actual, **length** devuelve el número de caracteres que forman la cadena actual, **Get** devuelve una subcadena, que comienza en la posición del primer parámetro y tiene tantos caracteres como indique el segundo. Se produce una excepción si la subcadena no está incluida en el string (inicio o longitud inadecuados).

Insert y **append** tienen un comportamiento similar. Mientras el primero inserta una subcadena en la posición indicada, produciendo una excepción si la cadena actual no llega a dicha posición, el segundo concatena, es decir, inserta al final de la cadena actual, la cadena pasada como parámetro.

Delete borra tantos caracteres como indique el segundo parámetro a partir de la posición que indique el primero. Se produce una excepción si el rango no es correcto.

GetChar devuelve un entero cuyo valor es el código ASCII del carácter que esté en la posición indicada. Se produce una excepción si el rango no es correcto. Por su parte, **SetChar**, establece como valor de la posición indicada en el primer parámetro el carácter cuyo código ASCII se corresponda con el valor indicado en el segundo parámetro. En caso de que el entero esté fuera del rango 0-255 se producirá una excepción, al igual que en el caso de que la posición indicada esté fuera de rango.

toUpper y **toLower** transforman todos los caracteres de la cadena a mayúsculas y minúsculas, respectivamente.

equal, **greater** y **less** establecen comparaciones entre el objeto actual y el indicado como parámetro, devolviendo un objeto de tipo bool. Concretamente, equal compara si ambas cadenas son iguales, greater si la cadena actual es mayor que la del parámetro y por último, less, compara si la cadena actual es menor que la del parámetro. Todas las comparaciones son sensibles al contexto.

1.6 Array

La clase array se limita a guardar las referencias de los objetos – no los propios objetos – en la posición que se le indique, para su posterior manipulación consulta.

No crea ni borra los objetos, sino que manipula sus referencias.

La interfaz que ofrece dicha clase es:

```
Class array
Isa object
Methods

   setSize(integer);
   getSize(): integer;
   setRef (integer, object);
   getRef (integer): object;

endclass
```

El método **setSize** se utiliza para indicar el tamaño del array. Se puede cambiar el tamaño del array en cualquier momento. Si éste se aumenta de n a m posiciones, las n primeras posiciones del nuevo array serán las mismas de antes. Si se disminuye, las posiciones que sobren simplemente se ignoran. Por su parte, **getSize**, simplemente devuelve el tamaño del array.

Los métodos **getRef** y **setRef** devuelven la referencia almacenada en una posición y almacenan una referencia en una posición, respectivamente. Ambos producen una excepción si el índice se sale fuera del rango.

1.7 Stream

Un stream es un depósito en donde se guardan y de donde se recuperan objetos. Es una clase virtual de la que se derivan el resto de los streams estándar de Carbayonia

Su interfaz es la que se muestra a continuación.

```
Class stream
Isa object
Methods

   write (object);
   read(): object;

EndClass
```

Los métodos **write** y **read** escriben o recuperan objetos del stream, respectivamente. Ambos producen una excepción si ocurre algún error en el proceso de lectura o escritura. El método read además producirá una excepción si el objeto a recuperar del stream no es compatible con la referencia utilizada (recuérdese que en el paso y retorno de valores se hace un cast dinámico que puede producir una excepción si los tipos no son compatibles).

1.8 ConStream

La clase conStream es un stream asociado con una consola, de tal manera que todo los que se escriba en él va a la pantalla y todo lo que de él se lea viene del teclado. Este stream permite la comunicación entre las aplicaciones y el usuario.

Su interfaz se muestra a continuación.

```
Class conStream
Isa stream
Methods

    write (object);
    read(): object;
    locate (x,y: integer);
    nextLine ();
    cls ();

EndClass
```

Los métodos **read** y **write**, se corresponden con la redefinición de los métodos del mismo nombre vistos en la clase anterior. El método **locate** sirve para situar la posición del cursor dentro de la pantalla. **NextLine** envía a la consola un salto de línea y **cls** produce el borrado de la pantalla.

1.9 FileStream

La clase FileStream representa un stream situado en un dispositivo de almacenamiento secundario, es decir, un fichero. A continuación se muestra su interfaz y se describen, brevemente, sus métodos.

```
Class fileStream
Isa stream
Methods

    open(string);
    close();
    isOpen(): bool;
    eof():bool;
    seek(integer);
    tell():integer;
    write(object);
    read():object;

endclass
```

La primera operación a realizar con un fileStream será invocar su método **open**. El parámetro de dicho método es un string que lo identifica en el almacenamiento secundario, es decir, su nombre. Siguiendo con la aproximación tradicional, al finalizar el trabajo con el stream deberá invocarse al método **close** para cerrar el fichero. El método **isOpen** devuelve un objeto de tipo bool que indica si el fichero está abierto.

Los métodos **read** y **write** son la redefinición de los métodos del mismo nombre de la clase stream.

El método **eof** devuelve un valor bool que indica si se está al final del fichero.

Seek sirve para posicionarse dentro del fichero, e indica en que posición del mismo (contado en objetos a partir del inicio del fichero) debe realizarse la próxima operación de entrada o salida. Por último, **Tell** devuelve un entero que indica la posición actual en la cual se realizará la próxima lectura o escritura.

1.10 Clase Lock

Clase básica para la gestión de la concurrencia. Su semántica es la habitual y su utilidad principal es hacer un control fino sobre las exclusiones a realizar en determinados servicios.

La interfaz de la clase es:

```
Class Lock
Isa object
Methods

    Set(integer);
    Acquire();
    Release();
    IsHeldbyCurrentThread(): bool;

Endclass
```

El método **Set** permite la inicialización de la cerradura al valor indicado como parámetro. Este sólo puede ser uno de dos valores, 1 si la cerradura está abierta, o 0, si está cerrada.

Los métodos **Acquire** y **Release**, corresponden a la adquisición y liberación de una cerradura por parte de un hilo durante la ejecución de un método. La semántica de ambos es la habitual. Si la cerradura que se intenta adquirir está cerrada, se suspende la ejecución del hilo hasta que aquel que la haya cerrado ejecute Release. En caso contrario, si la cerradura está abierta, el hilo actual continúa la ejecución del método y cierra la cerradura.

Por último, IsHeldbyCurrentThread devuelve un bool que tendrá valor cierto si la cerradura está cerrada por el hilo actual y falso en otro caso.

Anexo C

Jerarquía de Clases Primitivas Reflectivas de la máquina

En este anexo se presenta la jerarquía básica de clases ofrecida por la máquina para implementar el módulo de reflectividad y que el programador podrá ir ampliando posteriormente. Todas estas clases, al igual que las definidas en el Anexo B y las que el usuario pueda ir definiendo posteriormente, se encuentran registradas en el área de clases, las clases básicas, de forma permanente.

Para su manipulación se utilizan las instrucciones definidas en el Anexo A.

1 Clases primitivas para la Reflectividad Estructural

1.1 Exposición de los elementos que forman la Arquitectura de la Máquina

1.1.1 Clase ClassArea

La clase ClassArea representa o expone el área de clases de la máquina abstracta, permitiendo su acceso y modificación.

Su interfaz es la siguiente:

```
Class ClassArea
Isa object
Methods

    GetClasses():array;
    IsaClass(string):bool;
    GetClass(string):_Class;
    NewClass(string): _Class;
    DupClass(string, string): _Class;

Endclass
```

El método **GetClasses** obtiene del área de clases los identificadores de todas las clases almacenadas en el área de clases. Estos nombres se almacenan en un array de objetos de tipo _Class para su posterior utilización.

Por su parte **GetClass** obtiene un objeto de tipo _Class que representa a la clase cuyo nombre se pasa como parámetros. Esto hace que las clases sean accesibles y manipulables mediante instrucciones normales de la máquina, como se verá más adelante.

IsaClass interroga al área de clases acerca de la existencia de una clase con el nombre que se pasa como parámetro. El retorno de un valor cierto, supondrá la existencia de tal clase. En caso de retornar falso, tal clase no está definida.

El método **NewClass**, almacena en el área de clases una nueva clase cuya referencia se pasa como parámetro. Y por último, el método **DupClass** duplica una clase ya existente, que tenga como nombre la cadena pasada como primer parámetro, creando una segunda clase, que tiene como nombre el segundo parámetro pero que se coloca como descendiente directa de la clase origen en la jerarquía de clases. Posteriormente, gracias a la exposición de las clases, podremos modificar cualquiera de las clases creadas a voluntad.

1.1.2 Clase InstanceArea

La clase InstanceArea representa o expone el área de instancias de la máquina abstracta, permitiendo su acceso y modificación.

Su interfaz es la siguiente:

```
Class InstanceArea
Isa object
Methods

    GetInstances():array;
    NewInstance(string):instance;
    InstanceofthisRef(reference):instance;
    SetLoadbyUser(string, integer);
    GetLoadbyUser(string):integer;

Endclass
```

El método **GetInstances** obtiene referencias a todas las instancias almacenadas en el área de instancias que, a su vez, se almacenarán en un array para su posterior utilización.

Por su parte **NewInstance** crea una nueva instancia del tipo indicado como parámetro. Devuelve una referencia a un objeto de tipo instance que podrá manipularse con las instrucciones normales de la máquina.

El método **InstanceofThisRef** inspecciona el área de instancias buscando el objeto ligado a la referencia que se pasa como parámetro. Como resultado, crea un objeto de tipo instancia para reflejar las características de la instancia hallada.

Por último, los métodos **GetLoadbyUser** y **SetLoadbyUser** permiten averiguar y establecer, respectivamente, la carga de objetos que el usuario identificado tiene permitido. Su principal utilidad radica en el caso de que se eleven demasiadas excepciones por falta de espacio en el área de instancias, en cuyo caso, pueden establecerse límites a la creación de instancias.

La exposición de las instancias permite la creación y modificación de las mismas en tiempo de ejecución lo que resultará particularmente útil para implantar la reflectividad del comportamiento.

1.1.3 Clase ReferenceArea

La clase ReferenceArea representa o expone el área de referencias de la máquina abstracta, permitiendo su acceso y modificación.

Su interfaz es la siguiente:

```
Class ReferenceArea
Isa object
Methods

    NewReference(string):reference;
    SetLoadbyUser(string, integer);
    GetLoadbyUser(string):integer;

Endclass
```

El método **NewReference** permite crear una referencia nueva en el área de referencias. El parámetro que se pasa es el nombre de la clase.

Por último, los métodos **GetLoadbyUser** y **SetLoadbyUser** permiten averiguar y establecer, respectivamente, la carga de referencias que el usuario identificado tiene permitido. Similares a los métodos del mismo nombre de la clase InstanceArea.

1.1.4 Clase ThreadArea

El área de hilos permite el acceso y modificación del área del mismo nombre definida en la arquitectura de referencia de la máquina, en el Capítulo XV.

Su interfaz es:

```
Class ThreadArea
Isa object
Methods

    HowMany():integer;
    MaxLoad(integer);

Endclass
```

El método **HowMany** devuelve el número de hilos en el área. Mientras que **MaxLoad** permite establecer un límite superior al número de hilos **activos** en la máquina, con el fin de distribuir la carga en varias máquinas, si fuese necesario.

1.2 Exposición de los elementos estructurales de la implantación en tiempo de ejecución de los objetos

1.2.1 Clase _Class

Esta clase, que expone precisamente diversos aspectos de la representación en tiempo de ejecución de las clases, permite el acceso y modificación de cualquier clase definida en el sistema.

Su interfaz es la siguiente:

```
Class _Class
Isa object
Methods

   GetName():string;
   GetInterfaz():array;
   GetParent():array;
   GetHierarchy():array;
   GetAggregates():array;
   GetAssociates():array;
   SetHierarchy(array);
   SetAggregates(array);
   SetAssociates(array);
   GetMethod(string):Method;

Endclass
```

El conjunto de métodos de la clase _Class están divididos en aquellos que permiten **interrogar a la clase** acerca de su estructura y aquellos que permiten **actuar** sobre la misma, modificándola en tiempo de ejecución.

Así entre los primeros se encuentran los siguientes: **GetName** permite interrogar a una clase acerca del nombre asignado en la definición. Obviamente, devuelve ese nombre como un string de caracteres. **GetInterfaz** obtiene un array con los nombres del conjunto de métodos que la clase interrogada implementa. **GetParent** inspecciona la clase en busca de los antecesores directos en la jerarquía de clases, mientras **GetHierarchy** realiza una inspección completa de la cadena de herencia de la clase. **GetAggregates** y **GetAssociates** obtienen las clases agregadas y asociadas, respectivamente, y por último, el método **GetMethod** permite obtener la descripción de un método de la clase a partir de un string con su nombre.

Y entre los segundos, aquellos que permiten definir o modificar los elementos que constituyen la clase: **SetHierarchy**, **SetAggregates** o **SetAssociates**.

1.2.2 Clase Instance

Representa una instancia en tiempo de ejecución. Su interfaz consta de los siguientes métodos:

```
Class Instance
Isa object
Methods

   GetClass():string;
   GetExecArea():ExecObjectArea;
   GetMetaSpace():MetaSpace;
   GetAggregates():array;
   GetAssociates():array;

Endclass
```

Todos sus métodos se dirigen a realizar introspección sobre la instancia en tiempo de ejecución.

El método **GetClass** devuelve el nombre de la clase de la que es instancia, **GetExecArea** devuelve un objeto de tipo **ExecObjectArea** que, como se verá representa el área de Ejecución de cada objeto, **GetMetaSpace**, devuelve un objeto de tipo **MetaSpace** que engloba al conjunto de metaobjetos que componen el entorno de un objeto base. Este método es el método básico para permitir el acceso y manipulación al metaespacio de un objeto en tiempo de ejecución.

Por último, **GetAggregates** y **GetAssociates** devuelve un array que contiene, respectivamente, los agregados y los asociados de la instancia.

1.2.3 Clase Method

La clase Method expone los métodos de una clase y permite el acceso y modificación a los mismos. Permite crear métodos en tiempo de ejecución y es fundamental para poder definir clases en tiempo de ejecución.

Su interfaz es:

```
Class Method
Isa object
Methods

   GetName():string;
   GetInstructionsSet():array;
   GetReturnType():string;
   GetReferences():array;
   GetInstances():array;
   GetParams():array;
   SetInstructionsSet(array);
   SetReturnType(string);
   SetReferences(array);
   SetInstances(array);
   SetParams(array);

Endclass
```

Nuevamente, los métodos están divididos en dos conjuntos simétricos. El primero, compuesto por los métodos **GetName**, **GetInstructionsSet**, **GetReturnType**, **GetReferences**, **GetInstances** y **GetParams**, permite inspeccionar un método obteniendo un string con su nombre, un array de instrucciones que constituyen el cuerpo

del método, el tipo del valor de retorno, si existe, y las Referencias, instancias y parámetros que el método recibe, respectivamente.

El segundo conjunto, formado por **SetInstructionsSet**, **SetReturnType**, **SetReferences**, **SetInstances** y **SetParams**, permite dar valor y modificar el conjunto de instrucciones de un método, su valor de retorno, referencias, instancias y parámetros, que recibe, respectivamente.

1.2.4 Clase Instruction

La clase Instruction permite la creación de nuevos objetos instrucción de alguno de los tipos de instrucción definidos. Es imprescindible si se pretenden definir nuevas clases en tiempo de ejecución y su interfaz es la siguiente:

```
Class Instruction
Isa object
Methods

   NewAssign(string, string);
   NewCall(string, string, array, string);
   NewDelete(string);
   NewExit();
   NewHandler(integer);
   NewNew(string);
   NewJmp(integer);
   NewJT(string, integer);
   NewJF(string, integer);
   NewJNull(string, integer);
   NewJNNull(string, integer);
   NewSend(string, string, array, string);
   NewThrow();

Endclass
```

Los métodos se refieren todos a la creación de instrucciones de los tipos que hay definidos en el procesador recibiendo como parámetros los argumentos que necesitan tales instrucciones.

Así, **NewAssign**, crea una instrucción de asignación y necesita los nombres de las referencias que entran en juego en la asignación.

NewCall, se refiere a una instrucción de invocación síncrona. Sus parámetros son el nombre de la referencia, el nombre del método, los parámetros del método y la referencia de retorno, si existe.

NewDelete, borra la referencia que tenga como nombre el argumento enviado.

NewExit crea una instrucción de finalización.

NewHandler, crea una instrucción de tipo handler y recibe como parámetro el desplazamiento, dentro del cuerpo del método, donde salta el control de flujo.

NewNew, crea un objeto de tipo instrucción que representa la instrucción New. Obviamente, como parámetro recibe el nombre de la referencia.

NewJmp, crea una instrucción de salto incondicional. El parámetro representa el offset donde continúa el flujo de ejecución.

NewJT, **NewJF**, **NewJNull** y **NewJNNull**, crean instrucciones de salto condicional. Reciben como parámetro, en primer lugar el nombre de la referencia a inspeccionar y en

segundo lugar, el desplazamiento donde continúa el flujo de ejecución, en caso de que la instrucción tome valor cierto.

NewSend, al igual que NewCall, es una instrucción de paso de mensajes, pero, en este caso, asíncrona. Los parámetros son los mismos, ya que solo cambia la semántica de la instrucción.

Por último, **NewThrow** crea una instrucción de emisión de una excepción. No lleva parámetros.

1.3 Exposición de la engine en tiempo de ejecución

1.3.1 Clase ExecObjectArea

La clase ExecObjectArea representa o expone el área de ejecución del objeto, es decir, permite inspeccionar qué métodos se están ejecutando en dicho objeto. Esto, junto con la exposición del objeto interno thread que se verá a continuación, constituyen una herramienta imprescindible para poder definir la planificación en el meta-nivel, a la vez que suponen una gran ayuda para las tareas de depurado.

Su interfaz es la siguiente:

```
Class ExecObjectArea
Isa object
Methods

    GetThreads():array;
    GetLoad():integer;
    SetLoad():integer;

Endclass
```

El método **GetThreads** obtiene referencias a objetos de tipo hilo que representan todos los métodos en ejecución en el objeto.

1.3.2 Clase Thread

La clase Thread representa o expone la ejecución de cada uno de los métodos de un objeto. Permite inspeccionar el estado del hilo, la cadena de llamadas que rodean al hilo, hacia delante y hacia atrás, así como realizar acciones como suspender el hilo actual, reanudar un hilo, etc.

Su interfaz es la siguiente:

```
Class Thread
Isa object
Methods

    GetState():string;
    GetContext():context;
    GetPreviousThread():Thread;
    GetNextThread():Thread;
    Kill();
    Suspend();
    Resume();
    Start();

Endclass
```

El método **GetState** informa acerca del estado actual del hilo referenciado.

GetContext permite obtener el contexto de ejecución actual del hilo. Junto con la clase Context permite navegar por la pila de ejecuciones de métodos en el objeto al que está asociado. Si a ello se unen **GetPreviousThread** y **GetNextThread** que permiten obtener referencias a objetos de tipo hilo que representan el hilo anterior y siguiente, se puede seguir sin problema, toda la cadena de llamadas a métodos que incluye al presente thread.

Por último, los métodos **Kill**, **Suspend**, **Resume** y **Start** permiten actuar sobre el objeto interno de la máquina y son básicos para la construcción de planificadores que, de otra forma no podrían actuar sobre los hilos de la máquina.

Se ocupa de matar, suspender, reanudar y comenzar la ejecución de un hilo, respectivamente.

1.3.3 Clase Context

Permite el acceso a un contexto de ejecución de la máquina, aunque no sea el contexto activo actualmente. Los métodos de su interfaz son:

```
Class Context
Isa object
Methods

    GetPreviousContext():Context;
    GetNextContext():Context;
    GetInstances():array;
    GetReferences():array;
    GetMethod():string;

Endclass
```

GetPreviousContext y **GetNextContext** permiten explorar la pila de contextos de un thread moviéndose hacia delante y hacia atrás en la misma.

El resto de los métodos permiten explorar un contexto determinado obteniendo las referencias e instancias con los métodos **GetInstances** y **GetReferences** respectivamente. Ambos devuelven arrays con las referencias a instancias y referencias del contexto.

Por último, **GetMethod** identifica el método en ejecución en el contexto dado.

2 Clases primitivas para la Reflectividad del Comportamiento

2.1 Clase MetaSpace

La clase MetaSpace representa los objetos del MetaEspacio en tiempo de ejecución, permitiendo conocer qué meta objetos lo componen y modificarlos, logrando así adaptación del entorno en tiempo de ejecución.

La Clase MetaSpace tiene una interfaz muy simple.

```
class MetaSpace
isa Object
methods

  GetMetaSpace(): array;
  GetMObyClass(string):MetaObject;
  AttachMObyClass(string, MetaObject);

Endclass
```

El método **GetMetaSpace** devuelve un array de metaobjetos que componen el meta espacio. Posteriormente, se podrán inspeccionar estos meta objetos.

El método **GetMObyClass** devuelve el meta objeto de la clase especificada, mientras **AttachMObyClass** especifica un nuevo meta objeto para exponer determinada característica del entorno del objeto.

2.2 La Clase MetaObjeto

La clase MetaObjeto representa los Meta Objetos en tiempo de ejecución. Se expresa en una jerarquía aparte de los objetos de usuario, aunque se codifican en Carbayón y pueden – y deben – ser definidos por el usuario.

La Clase MetaObjeto tiene una interfaz muy simple – igual que la clase objeto – y cada subclase de la misma especializará su comportamiento y creará una jerarquía independiente.

```
class MetaObject
isa Object
methods

  GetClass(): string;
  GetID():integer;
  Isa(string):bool;
  GetObjectBase():array;

Endclass
```

El método **getClass** devuelve un string (clase que se verá posteriormente) que contiene el nombre de la clase a la que pertenece el objeto.

El método **getID** devuelve una instancia de un entero que identifica de forma única al objeto. De esta forma se puede averiguar si se tiene dos referencias apuntando al mismo objeto (aunque estas sean de distinto tipo).

El tercer método –**isa**– devuelve una instancia de un objeto de tipo bool que indica si la instancia pertenece a esa clase o a una derivada de ella.

Por último, **GetObjectBase** devuelve una referencia a un array de objetos de cuyo metaespacio forma parte.

A continuación se ofrece la jerarquía básica de metaobjetos. Cada usuario puede, no sólo modificar los métodos de la interfaz, sino también añadir nuevos métodos, ampliando de esta forma su jerarquía.

2.3 Clase Emisor

La clase Emisor representa los objetos del MetaEspacio que se encargan del envío de mensajes en tiempo de ejecución.

La interfaz base de esta clase es tan simple como sigue:

```
class MOEmisor
isa MetaObject
methods

  RCallE(string, string, array, string): object;

Endclass
```

El método **RCallE** define las operaciones de envío de mensajes. Los parámetros que recibe son el nombre de la referencia del objeto destino, el nombre del método a invocar, un array con los argumentos y el nombre de la referencia donde se guarda el resultado, si existe.

Este método retorna diversos resultado, en función de cómo esté configurado el metaobjeto destino.

Este resultado puede utilizarse para ampliar la semántica de este meobjeto.

Un ejemplo de metaobjeto Emisor más sofisticado sería el siguiente:

```
class MOEmisorComplex
isa MOEmisor
methods

  RCallE(string, string, array, string, integer): object;
  MethodTimeOut();
  NoFindDest();

Endclass
```

Este metaobjeto redefine el metaobjeto anterior modificando el método **RCallE** de tal forma que, entre las operaciones de envío de mensajes, se instale una alarma que

controle el tiempo de ejecución de un método. También define un comportamiento excepcional en caso de que no se encuentre el objeto destino.

Los parámetros que recibe son los mismos, si exceptuamos el número de segundos que se intenta la ejecución del método.

El método **MethodTimeOut** define las acciones a realizar en caso de que el objeto destino no sea capaz de ejecutar el método solicitado en el tiempo especificado.

El método **NoFindDest** especifica las acciones a realizar en caso de que no se encuentre el objeto destino. La acción por defecto es elevar una excepción. Sin embargo, puede intentar solucionarse buscando un servidor alternativo.

2.4 Clase Receptor

La clase Receptor representa los objetos del MetaEspacio que se encargan de la recepción de mensajes en tiempo de ejecución.

La interfaz base de esta clase es tan simple como sigue:

```
class MOReceptor
isa MetaObject
methods

  RCallR(string, string, array, string): object;

Endclass
```

El método **RCallR** define las operaciones de recepción de mensajes. Los parámetros que recibe son el nombre de la referencia del objeto origen, el nombre del método a invocar, un array con los argumentos y el nombre de la referencia donde se guarda el resultado, si existe.

Este método retorna diversos resultado, en función de cómo esté configurado el metaobjeto destino y los retornará al metaobjeto Emisor.

Un ejemplo de metaobjeto Receptor más sofisticado sería el siguiente:

```
class MOReceptorComplex
isa MOReceptor
methods

  RCallR(string, string, array, string): object;
  TooLoad(string, string, array, string): object;
  DelegateMethod(string, string, array, string): object;
  DelayMehod();

Endclass
```

Este metaobjeto redefine el metaobjeto anterior modificando el método **RCallR** de tal forma que, entre las operaciones de recepción de mensajes, se estudie el entorno en tiempo de ejecución del objeto base tomándose decisiones acerca de la posibilidad de ejecución del mismo en función de la carga.

TooLoad define las acciones a realizar en caso de carga excesiva, por defecto el método RCallR rechazaría el mensaje. Los parámetros son, nuevamente, el nombre de la referencia a quién delegar, el nombre del método, los argumentos y la referencia de retorno. Así mismo, devuelve un resultado que dependerá de si todo ha ido bien o si se ha producido algún problema.

Otras acciones pueden ser delegarlo en otro objeto (posiblemente una réplica), **DelegateMethod** o retrasar su ejecución, **DelayMethod**.

2.5 Clase Sincronizador

La clase Sincronizador representa los objetos del MetaEspacio que se encargan de definir el comportamiento del objeto ante la invocación de métodos.

La interfaz base de esta clase es tan simple como sigue:

```
class MOSynchronizer
isa MetaObject
methods

  ExecNewMethod(string, string): object;
  EndMethod(string);
  RestartMethod(string);
  StopMethod(string);

Endclass
```

El método **ExecNewMethod** indica las acciones a realizar cuando se solicita la ejecución de un método. Los parámetros corresponden al origen de la llamada y al nombre del método, con el fin de evitar bloqueos mutuos.

Los métodos **EndMethod** y **StopMethod** definen las operaciones que se llevan a cabo para determinar si otros métodos pueden pasar a ejecutarse como consecuencia de la finalización o suspensión del método cuyo nombre se pasa como parámetro.

De forma similar el método **RestartMethod** implica la reanudación de la ejecución del método, lo que puede implicar que otros cambien su estado.

La acción por defecto es la más liberal posible lo que implica que cualquier método puede ejecutarse siempre, dejando en manos del programador la sincronización adecuada. Los métodos estarían vacíos, excepto por la llamada al planificador correspondiente para advertirle de la presencia de un nuevo hilo de ejecución.

Pero cada usuario puede redefinir estos métodos aplicando la política de sincronización que desee.

Un ejemplo de Sincronizador más sofisticado, sería aquel que definiese una actuación distinta para cada método:

```
class MOSynchronizerComplex
isa MetaObject
methods

  ExecNewMethod(string, string): object;
  EndMethod(string);
```

```
    RestartMethod(string);
    StopMethod(string);
    EntryM();
    ExitM();
    StopM();
    ResumeM();
    IentryM();
    IexitM();
    IstopM();
    IresumeM();

Endclass
```

Donde los métodos EntryM, ExitM, StopM, ResumeM, IentryM, IexitM, IstopM, IresumeM estarían definidos para todos o parte de los métodos M del objeto del nivel base.

2.6 Clase Scheduler

Por último, la clase Scheduler representa los objetos del MetaEspacio que se encargan de definir la planificación del entorno.

La interfaz base de la clase de planificador universal es tan simple como sigue:

```
class MOUniversalScheduler
aggregation queue
isa MetaObject
methods

    ScheduleNext();
    Enqueue(Thread);
    IsEmpty():Bool;
    GetQueue():queue;

Endclass
```

El método **ScheduleNext** indica a la máquina abstracta la referencia al hilo del que debe ejecutar instrucciones.

Enqueue inserta en la cola de hilos un nuevo hilo en la posición indicada. Y, por último, **IsEmpty** inspecciona la cola de hilos para determinar si hay alguno y GetQueue retorna una cola con los hilos que este planificador tenga definidos.

Referencias

[AA99]	F.Álvarez García y D.Álvarez Gutiérrez. "Component Object Model (COM)". En *Handbook of Object Technology* , pág. 32-1 a 32-17.Saba Zamir, ed. CRC Press. 1999.
[AAS+98]	G.Agha, M.Astley, J.Sheikh y C.Varela. "Modular Heterogeneous System Development: A Critical Analysis of Java". En *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW '98)*, 1998.
[ABB+93]	H.Assenmacher, T.Breitbach, P.Buhler, V.Huebsch y R. Schwarz. "The PANDA system architecture - a pico-kernel approach". En *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*, Pág 470-476. IEEE Computer Society Press, Septiembre 1993.
[ABG+86]	M. Acceta, R. Baron, D. Golub, R. Rashid, A. Tevanian y M. Young. "Mach: A New Kernel Foundation for Unix Development". En Proceedings of the *Summer Usenix Conference*. USENIX. 1986. Pág. 93-112.
[ABL+85]	G.T.Almes, A.P.Black, E.D.Lazowska y J.D.Noe. "The Eden System: A Technical Review". *IEEE Transactions on Software Engineering*, SE-11(1):43-58. 1985
[ABL+91]	T.E.Anderson, B.N.Bershad, E.D.Lazowska y H.M. Levy. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism". En *Proceedings of the Thirteenth Symposium on Operating System Principles*. Pp. 95-109. 1991
[AG97]	K.Arnold y J.Gosling. *The Java Programming Language. 2nd Edition*. Addison-Wesley. 1997
[Agh86]	G.Agha. *Actors: A Model of Cocnurrent Computation in Distributed Systems.* MIT Press, 1986
[AK95]	J.M.Assumpcao Jr y S.T.Kofuji. "Bootstrapping the Object Oriented Operating System Merlin: Just Add Reflection". En *Proceedings of the European Conference on Object Oriented Programming(ECOOP'95), Meta'95 Workshop on Advances in Metaobject Protocols and Reflection*. 1995
[AK98]	G.A.Agha y W.Kim, "Actors: a unifying model for parallel and distributed computing". *Journal of Systems Architecture. Special Issue on New Trends in Programming and Execution Models for Parallel Architectures, Heterogeneously Distributed Systems and Mobile Computing*. EuroMicro Society en cooperación con Elsevier. 1998
[ALK+90]	A.Agarwal, B.-H.Lim, D.Kranz y J.Kubiatowicz. "APRIL: Aprocessor Architecture for Multiprocessing". En Proceedings of the 17th Annual Symposium on Computer Architecture. Pp. 104-114. Mayo 1990.
[Álv97]	D.Álvarez Gutiérrez. Tesis.

[Ame88]	P.America. "POOL-T: A Parallel Object-Oriented Language". En [YT88b].
[And92]	T.Anderson. "The case for application-specific operating systems". En *Proceedings of the 3rd Workshop on Workstation Operating Systems*, 92-94. IEEE Computer Society, IEEE Computer Society Press, 1992
[AR84]	W.F. Appelbe y A.P. Ravn. "Encapsulation Constructs in Systems Programming Languages". *ACM Transactions on Programming Languages and Systems*, 6(2):129-158, Abril 1984
[ATA+98]	D.Álvarez Gutiérrez, L.Tajes Martínez, F.Álvarez García, M.A.Díaz Fondón, R.Izquierdo Castanedo y J.M.Cueva Lovelle. "An Object-Oriented Abstract Machine as the Substrate for an Object-Oriented Operating System". En *11th European Conference on Object-Oriented Programming (ECOOP'97). Workshop on Object-Orientation and Operating Systems.* Object-Oriented Technology. Lecture Notes in Computer Science 1357, pág. 537-544. I.Bosch, y S. Mitchell (Eds.) Springer-Verlag, 1998
[ATD+98a]	F.Álvarez García, L.Tajes Martínez, M.A.Díaz Fondón, D.Álvarez Gutiérrez y J.M.Cueva Lovelle. "Agra: The Object Distribution Subsystem of the SO4 Object-Oriented Operating System". En *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98).* H.R.Arabnia, ed. pp. 255-258. CSREA Press. 1998
[ATD+98b]	F.Álvarez García, L.Tajes Martínez, M.A.Díaz Fondón, D.Álvarez Gutiérrez y J.M.Cueva Lovelle. "Object Distribution Subsystem for an Integral Object-Oriented System". En *Anales del Simposio Argentino en Orientación a Objetos(ASOO'98).* Pp. 219-227. Buenos Aires, Argentina, Octubre de 1998
[ATT91]	AT&T. *Unix System V: Interface Definition.* 1991
[~SOM]	*System Object Model.* URL: http://info.gte.com/ftp/doc/activities/x3h7/by_model/SOM.html
[~Tun]	The Tunes Glossary. URL: http://www.tunes.org
[Bac86]	M.J.Bach. *The Design of the UNIX Operating System*. Prentice Hall, Englewood Cliffs, 1986.
[Bac92]	Jean Bacon. *Concurrent Systems. An integrated approach to Operating Systems, Database, and Distributed Systems.* Addison-Wesley, 1992
[BBD+91]	R.Balter, J.Bernadat, D.Decouchant, A.Duda, A.Freyssinet, S.Krakowiak, M.Meysembourg, P.Le Dot, H.Nguyen Van, E. Paire, M.Riveill, C.Roisin, X.Rousset de Pina, R.Scioville y G.Vandôme. "Architecture and Implementation of Guide, an Object-Oriented Distributed System". *Computing Systems*, 4(1):31-67. 1991
[BC87]	J-P.Briot y P.Cointe. "A Uniform Model for Object-Oriented Languages Using the Class Abstraction". IJCAI'87. Agosto, 1987.
[BC89]	J-P.Briot y P.Cointe. "Programming with Explicit Metaclasses in Smalltalk-80".En *Proceedings of the Object-Oriented Programming,*

	Languages, Systems and Applications (OOPSLA'89). SIGPLAN Notices, 24:419-431, Octubre 1989
[BCC+91]	F.Boyer, J.Cayuela, P.-Y.Chevalier y D.Hagimont. "Supporting an Object-Oriented Distributed System: experience with Unix, Chorus and Mach". En *Proceedings of the Second Symposium on Experience with Distributed and Multiprocessor Systems(SEDMS-2)*, 1991
[BCE+94]	B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage y E. Gün Sirer. "SPIN – An Extensible Microkernel for Application-specific Operating System Services". En *Proceedings of the 1994 European SIGOPS Workshop*. Septiembre, 1994.
[BDM+73]	G. Birtwistle, O. Dahl, B. Mhyrtag y K. Nygaard. *Simula Begin*. Auerbach Press, 1973.
[Ben82]	M.Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall International. 1982
[Ben90]	M.Ben-Ari. *Principles of Concurrent and Distributed Programming*. 2nd edition. Prentice-Hall International Series in Computer Science. 1990
[BG96]	J.-P.Briot y R.Guerraoui. "A Classification of Various Approaches for Object-Based Parallel and Distributed Programming". Technical Report École Polytechnique Fédérale de Lausanne (EPFL) y Universidad de Tokio. 1996.
[BGL98]	J.P.Briot, R.Guerraoui y K.P.Löhr. "Concurrency and Distribution in Object-Oriented Programming". *ACM Computing Surveys*, Septiembre 1998.
[BGW93]	D.G.Bobrow, R.G.Gabriel y J.L.White. *CLOS in Context-The Shape of the Design Space*. En Object Oriented Programming – The CLOS Perspective. MIT Press, 1993
[BHJ+86]	A.Black, N.Hutchinson, E.Jul y H.Levi. "Object Structure in the Emerald System". En *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*. ACM, 1986.
[BHJ+87]	A.Black, N.Hutchinson, E.Jul, H.Levi y L.Carter. "Distribution and Abstract Data Types in Emerald". *Transactions on Software Engineering*, SE-13(1):65-76.
[Bir91]	A.Birrell. *An introduction to programming with threads*. En Systems Programming with Modula-3. G.Nelson, ed. Prentice-Hall. 1991.
[BKT89]	H.E.Bal, M.F.Kaashoek y A.S.Tanenbaum. "A Distributed Implementation of the Shared Data-Object Model". En *Workshop Proceedings of Distributed and Multiprocessor Systems*, pp. 1-19. USENIX Association, 1989.
[Bla90]	D.Black. "Scheduling Support for Concurrency and Parallelism in the Mach Operating System". *IEEE Computer Magazine*, 23(5):35-43. Mayo 1990.
[BLR94]	R. Balter, S. Lacourte y M. Riveill. "The Guide language". *The Computer Journal*. 1994

[BLR96]	M.N.Bouraqadi-Saâdani, T.Ledoux y F.Rivard. "Metaclass Composability". En *Proceedings of Composability Workshop. 10th European Conference on Object-Oriented Programming (ECOOP'96)*, Springer-Verlag, 1996.
[Boo94]	G.Booch. *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Co. Inc, 1994
[Box98]	D.Box. *Essential COM*. Object Technology Series. Addison-Wesley. 1998.
[Bri89]	J.P.Briot. "Actalk: A testbed for Classifying and Designing Actor Languages in Smalltalk-89". En *Proceedings of ECOOP'89*, Nottingham, Julio 1989.
[BS88]	L.Bic y A.C.Shaw. *The Logical Design of Operating Systems.* 2nd Edition. Prentice-Hall, 1988
[BSP+95]	B. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M.Fiuczynski, D.Becker, S. Eggers y C. Chambers. "Extensibility, safety and performance in the SPIN Operating System". En *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles.* 1995.
[Byt95]	Byte. "Object Babel". *Special Report OO Meets OS, BYTE*. 1995
[Cah96a]	V.Cahill. "Flexibility in Object-Oriented Operating Systems: A Review". *3rd CaberNet Radicals Workshop*. 1996
[Cah96b]	V.Cahill. "An Overview of the Tigger Object-Support Operating System Framework". En K.Jeffery, J.Král y M. Barlosek, eds. *SOFSEM'96: Theory and Practice of Informatics*. Vol 1175 of Lecture Notes in Computer-Science:34-45. Springer-Verlag, 1996
[Car89]	D.Caromel. "Service, Asynchrony and Wait-by-Necessity". *Journal of Object Oriented Programming (JOOP)*, Pág. 12-22, November, 1989
[Car93]	D.Caromel. "Towards a Method of Object-Oriented Concurrent Programming". *Communications of the ACM*, 36(9):90-102. September, 1993
[Caz98]	W.Cazzola. "Evaluation of Object-Oriented Reflective Models". En *Object-Oriented Technology. ECOOP'98 Workshop Reader*. S.Demeyer y J.Bosch, eds. Pág 386-387. Lecture Notes in Computer Science 1543. Springer-Verlag, 1998.
[CC91]	R. S. Chin y S. T. Chanson. "Distributed Object-Based Programming Systems". *ACM Computing Surveys*, 23(1). March 1991. Pág. 91-124.
[CD88]	E.C.Cooper y R.P.Draves. "C threads". Tecnical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University. 1988.
[CD94]	D.Cheriton y K.Duda. "A Caching Model of Operating System Kernel Functionality". *Proceedings of OSDI*. Pp. 179-193, 1994
[CFH+93]	P.Y.Chevalier, A.Freyssinet, D.Hagimont, S.Krakowiak, S.Lacourte y X.Rousset de Pina. "Experience with Shared Object Support in the Guide System".

	➔ **Datos publicación**
[Cha92]	C.Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD Thesis. Stanford University. Marzo, 1992
[Che84]	D.Cheriton."The V Kernel: A Software Base for Distributed Systems". IEEE Software, 1(2):19-42. 1984
[Che88]	D.Cheriton."The V Distributed System". *Communications of the ACM*, 31(3):314-333. 1988
[CIA+97]	
[CIM92]	R.H. Campbell, N. Islam y P.W. Madany. "*Choices*, Frameworks and Refinement". *Computing Systems*, 5(3):217-257, 1992
[CIM+93]	R.H. Campbell, N. Islam, P.W. Madany y D. Raila. "Designing and Implementing Choices: an Object-Oriented Operating System in C++". *Communications of the ACM*, Septiembre 1993.
[CJM+89]	R.H. Campbell, G.M. Johnston, P.W. Madany y V.F. Russo. "Principles of Object-Oriented Operating Systems Design". *Technical Report UIUCDCS-R-89-1510, TTR89-14*, University of Illinois at Urbana-Champaign, Department of Computer Science, 1989
[CL95]	M. Cheung y A. Loong. "Exploring Issues of Operating Systems Structuring: From Microkernel to Extensible Systems". *Operating Systems Review*, vol 29, nº 4, pág 4-16, 1995.
[Coi87]	P.Cointe. "MetaClasses are first class objects: the ObjVLisp model". En *Proceedings of OOPSLA'87*, SIGPLAN Notices, Vol. 22, ACM, Orlando, Florida, Octubre 1987
[CPR+92]	M.Chereque, D.Powell, P.Reynier, J.-L.Richier y J.Voiron. "Active Replication in Delta-4.22". *International Symposium on Fault-Tolerant Computing*, Pp. 28-37, IEEE, 1992
[Cro96]	C. Crowley. *Operating Systems. A Design-Oriented Approach*. McGraw-Hill. 1996
[CS91]	J.Cordsen, W.Schröder-Preikschat. "Object-Oriented System Design and the Revival of program families". En *Proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems (IWOOOS'91)*. Pag. 24-28. L.F.Cabrera, V.Russo y M.Shapiro, eds. IEEE Computer Society, IEEE Computer Society Press, 1991.
[CS98]	J.L.Contreras y J.-L.Sourrouille. "Adaptive Active Object". En *Object-Oriented Technology. ECOOP'98 Workshop Reader*. S.Demeyer y J.Bosch, eds. Pág 369-371. Lecture Notes in Computer Science 1543. Springer-Verlag, 1998.
[CV98]	D.Caromel y J. Vayssière "A Java Framework for Seamless Sequential, Multi-Threaded, and Distributed Programming". En *Proceedings of the*

	ACM Workshop Java for High-Performance Network Computing. Pag. 141-150. 1998.
[DAT+98]	M.A.Díaz Fondón, D.Álvarez Gutiérrez, L.Tajes Martínez, F.Álvarez García y J.M.Cueva Lovelle. "Capability-Based Protection for Integral Object-Oriented Systems". En *Proceedings of the COMPSAC'98, 22nd Conference in Computer Software and Applications*, pp. 344-349. IEEE Computer Society Press, 1998.
[DKM+89]	D.Decouchant, S.Krakowiak, M.Meysembourg, M.Riveill y R.de Pina. "A Synchronization Mechanism for Typed Objects in a Distributed System". *SIGPLAN Notices*, 24(4). Abril, 1989.
[DLA+91]	P.Dasgupta, R.J.LeBlanc, M.Ahamad y U. Ramachandran. "The Clouds Distributed Operating System". *IEEE Computer*, 24(11):34-44. 1991
[Don90]	C. Dony. "Exception Handling and Object-Oriented Programming: Towards a synthesis". *Proceedings of the OOPSLA'90*. 1990
[Dra93]	R.P.Draves. "The case for run-time replaceable kernel modules" En *Proceedings of the 4th Workshop on Workstation Operating Systems.* IEEE Computer Society Press, October 1993
[dRS84]	J.des Rivieres y B.C.Smith. "The implementation of procedurally reflective languages". En *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*. Pág. 331-347. Agosto 1984.
[EK95]	D.W.Engler y M.F.Kaashoek. "Exterminate All Operating System Abstractions". 5th Workshop on Hot Topics in Operating Systems (HotOS-V). Pp. 78-83 IEEE Computer Society Press. 1995
[EKT94]	D. R. Engler, M. F. Kaashoek y J. O'Toole Jr. "The Operating System Kernel as a secure Programmable machine". En *Proceedings of the Sixth SIGOPS European Workshop*. Pag. 62-67. 1994.
[EKT95]	D. R. Engler, M. F. Kaashoek y J. O'Toole Jr. "ExoKernel: An Operating System Architecture for Application-Level Resource Management". En *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. Pág 251-266. 1995.
[Exp95]	Expertsoft. *Expertsoft Xshell User's Manual Volume One*, *Release 3.5*, Marzo 1995
[Fer88]	J. Ferber. "Conceptual Reflection and actor Languages". En *Meta-Level Architectures and Reflection*, P. Maes, D. Nardi (eds.) Elsevier Science Publishers B.V. (North-Holland), 1988
[Fer89]	J.Ferber."Computational Reflection in Class Based Object Oriented Languages". En *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'1989).* SIGPLAN Notices, 24:317-326, ACM 1989.
[FHG93]	R.Finlayson, M.Hannecke y S.Goldbert. "From V to Vanguard: The Evolution of a Distributed, Object-Oriented Microkernel Interface". En *Proceedings of the USENIX Symposium on Microkernels and other Kernel*

	Architectures. Pág 27-30, 1993
[FJ89]	B.Foote y R.E.Johnson. "Reflective Facilities in Smalltalk-80".En *Proceedings of the Object-Oriented Programming, Languages, Systems and Applications (OOPSLA'89).* ACM, 1989
[Fla97]	D.Flanagan. *Java in a Nutshell. A Desktop Quick Reference for Java Programmers, 2nd edition*. O'Reilly & Associates. 1997
[Foo90]	B.Foote. " Object-Oriented Reflective Metalevel Architectures: Pyrite or Panacea?". En *ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures.*
[FS96]	B.Ford y S. Susarla. 'CPU Inheritance Scheduling". En *Proceedings of the OSDI'96*. 1996
[GC95]	B.Gowing y V.Cahill. "Making Meta-Object Protocols Practical for Operating Systems". En *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems (IWOOOS'95)*, Lind, Suecia, 1995.
[GC96]	B.Gowing y V.Cahill. "Meta-Object Protocols for C++: The Iguana Approach". En Proceedings of the Reflection'96. Pp. 137-152. Abril 1996.
[Gie90]	M.Gien. "Micro-Kernel Architecture Key to Modern Operating Systems Design". *UNIX REVIEW.* 8(11). Noviembre, 1990.
[GIL+95]	A.Gopal, N.Islan, B.-H.Lim y B.Mukherjee. "Structuring Operating Systems using Adaptive Objects for Improving Performance". En *Proceedings of the Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS'95).* Pp. 130-133. IEEE Computer Society, IEEE Computer Society Press, 1995
[GJS96]	J.Gosling, B.Joy y G.Steele. *The Java Language Specification*. Addison-Wesley. 1996
[GK97a]	M.Golm y J.Kleinöder. "MetaJava – A Platform for Adaptable Operating-System Mechanisms" **En el ecooop'97** .
[GK97b]	M. Golm y J. Kleinöder. "Implementing Real-Time Actors with MetaJava". En *ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Programming and Systems*, Junio 1997, Jyväskylä, Finlandia
[GKS94]	S. Graupner, W. Kalfa y F. Schubert. "Multi-level Architecture of object-oriented Operating Systems". *Technical Report TR-94-056, International Computer Science Institute (ICSI)*, 1994
[GM96]	J.Gosling y H.McGilton. *The Java Language Environment*. Sun Microsystems Computer Company. Mayo 1996.
[Golm97]	M. Golm. "Design and Implementation of a Meta Architecture for Java". Diplomarbeit im Fach Informatik. 1997

[GR83]	A.Goldberg y D.Robson. *Smalltalk-80. The Language and its Implementation*. Addison-Wesley. 1983.
[Gra89]	N.Graube. "MetaClass Compatibility". En *Proceedings of OOPSLA'89*, SIGPLAN Notices, 24:305-316, ACM. New Orleans, Lousiana, Octubre 1989
[GW96]	Brendan Gowing y Vinny Cahill. "Reflection + Micro-Kernels: An Approach to Supporting Dynamically Adaptable System Services". *CaberNet Radicals '96*. Connemara, Irlanda. 1996.
[Hau93]	F.J.Hauck. "Towards the implementation of a uniform object model". *Parallel Computer Architectures: Theory, Hardware, Software and Applications*. A.Bode y H.Wedekind, eds. Lecture Notes in Computer Science
[HFC76]	A.Habermann, L.Flon y L.Cooprider. "Modularization and Hierarchy in a Family of Operating Systems". *Communications of the ACM*, 19(5):266-272, 1976
[HK93]	G.Hamilton y P.Kougiouris. "The Spring Nucleus: A microkernel for objects". En *Proceedings of the 1993 Summer USENIX Conference*. Junio 1993.
[HKN91]	G.Hamilton, Y.Khalidi y M.Nelson. "Why Object oriented operating systems are boring". En *Proceedings of the 1st International Workshop on Object-Orientation in Operating Systems.* Pág 118-119. L.Cabrera , V.Russo y M.Shapiro, eds. IEEE Computer Society, IEEE Computer Society Press, octubre 1991.
[HMA90]	S.Habert, L.Mosseri y V.Abrossimov. "COOL: Kernel Support for Object-Oriented Environment". En *ECOOP/OOPSLA'90 Conference. SIGPLAN Notices*, 25: 269-277, ACM Press, 1990
[Hof99]	M.Hof. "Object Model with Exchangeable Invocation Semantics". En *2nd ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'99).* Lisboa, Portugal, Junio 1999.
[HPM+97]	M.Horie, J.C.Pang, E.G.Manning y G.C.Shoja. "Designing Meta-Interfaces for Object-Oriented Operating Systems". En *Proceedings of the 1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing-PACRIM'97*. IEEE Computer Society Press, 1997
[Hut87]	N.Hutchinson. "Emerald: An Object-Based Language for Distributed Programming". *Technical Report 87-01-01, University of Washington*, 1987.
[IBM96a]	IBM Corporation, *SOMobjects Developer´ Toolkit. Version 3.0. Programming Guide*, Vol. 1 y 2. 1996
[IBM96b]	IBM Corporation, *SOMobjects Developer´ Toolkit. Version 3.0. Programming Reference*, Vol. 1 a 4. 1996
[IYT95]	J.Itoh, Y.Yokote y M.Tokoro. "SCONE: Using Concurrent Objects for Low-Level Operating Systems Programming". En *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'1995).*ACM 1995.

[Izq96]	R.Izquierdo Castanedo. "". PFC.
[Izq99]	R.Izquierdo Castanedo. "Distributed System Object Model (DSOM)". En *Handbook of Object Technology*, pág. 31-1 a 31-15. Saba Zamir, ed. CRC Press. 1999.
[JBV93]	W.Joosen, S.Bijnens y P.Verbaeten. "Massively Parallel Programming using Object Parallelism". En *Proceedings of the Conference on Massively Parallel Programming Paradigms.* W.Giloi, S.Jahnichen y B.Shriver, eds. Pág 144-151. IEEE Computer Society Press, 1993.
[JLH+88]	Eric Jul, Henry Levy, Norman Hutchinson y Andrew Black. "Fine-Grained Mobility in the Emerald System". *ACM Transactions on Computer Systems*, 6(1):109-133, 1988
[JR86]	M.B.Jones y R.F.Rashid. "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems". En *Proceedings of First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).* 1986.
[KB93]	M.Karaorman y J.Bruno. "Introducing Concurrency to a Sequential Language". En [Mey93], Workshop on Efficient Implementation of Concurrent Object-Oriented Languages, pp. 103-116.
[KG96]	J.Kleinöder y M.Golm. "MetaJava: An Efficient Run-Time Meta Architecture for Java". En *Proceedings of the International Workshop on Object Orientation in Operating Systems – IWOOOS'96*. IEEE Computer Society Press. 1996.
[Kic92]	G.Kiczales. "Towards a New Model of Abstraction in Software Engineering". En *Proceedings of the International Workshop on New Models for Software Architecture'92; Reflection and Meta-Level Architectures*. A. Yonezawa y B.C.Smith, Eds. Pág 1-11, Tokyo, Japón, 1992
[Kic96]	G.Kiczales. "Beyond the black box: open implementation". IEEE Software, 13(1):8-11, Enero 1996
[KL93]	G. Kiczales y J. Lamping. "Operating Systems: Why Object-Oriented?". En Proceedings of the *Third International Workshop on Object-Orientation in Operating Systems.* L.F. Cabrera y N. Hutchinson, eds. IEEE Computer Society Press, 1993. Pág. 25-30
[KLM+93]	G. Kiczales, J. Lamping, C.Maeda, D.Keppel y D.McNamee. "The Need for Customizable Operating Systems". En Proceedings of the *Fourth Workshop on Workstation Operating Systems.* Pp. 165-169. IEEE Computer Society Technical Committee on Operating Systems and Application Environments. IEEE Computer Society Press, 1993.
[KNP88]	D.Konstantas, O.Nierstrasz y M.Papathomas. "An Implementation of Hybrid, a Concurrent Object-Oriented Language". *En Active Object Environments*, D. Tsichritzis (Ed.), pág 61-105. Centre Universitaire d'Informatique, University of Geneva, Junio 1988
[Kon93]	D. Konstantas. "Object Oriented Interoperability". En Proceedings of the *Seventh European Conference on Object Oriented Programming (ECOOP93)*. 1993

[KR94]	J.Kleinöder y T.Riechmann. "Hierarchical Schedulers in the PM System-Architecture". *Technical Report TR-I4-94-16. University of Erlangen-Nürnberg*.
[Kra93]	S. Krakowiak. "Issues in Object-Oriented Distributed Systems". En *Proceedings of the International Conference on Decentralized and Distributed Systems. IFIP WB 10.3*, 1993. Pág. 1-12
[KRB91]	G.Kickzales, J. desRivières y D.G.Bobrow. *The art of MetaObject Protocol*. MIT Press, Cambridge, Massachusetts, 1991
[KTW92]	G.Kiczales, M.Theimer y B.Welch. "A new model of abstraction for operating system design". En *Proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems.* Pág 346-349. L.-F.Cabrera y E. Jul, eds. IEEE Computer Society, IEEE Computer Society Press, septiembre 1992.
[LCJ+87]	B.Liskov, D.Curtis, P.Johnson y R.Scheifler. "Implementation of Argus". En *ACM Proceedings of the 12th Symposium on Operating System Principles*. Pp. 111-122.
[Lie95]	J.Liedtke. "On μ-kernel Construction". En *SIGOPS '95. Proceedings of the fifteenth ACM symposium on Operating systems principles*. ACM SIGOPS Operating Systems Review. 29(5): 237-250. 1995
[Lin95]	A.Lindström. "Multiversioning and Logging in the Grasshopper Kernel Persistent Store". En *Prodeedings of the 4th International Workshop on Object Orientation in Operating Systems.* Pp. 14-23 IEEE, 1995
[LP90]	W.R.Lalonde y J.R.Pugh. *Inside Smalltalk (Volume 1)*. Prentice-Hall International editions. Englewood Cliffs, 1990.
[LS82]	B.Liskov y R.Scheiffer. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs". *9th ACM Symposium on Principles of Programming Languages*. 1982.
[LJP93]	R.Lea, C.Jacquemot y E.Pillenvesse. "COOL: system support for distributed object-oriented programming". *Communications of the ACM, Special Issue on Concurrent Object-Oriented Programmings*, 36(9). 1993
[LTS+96]	Y.Li, S.-M.Tan, M.L.Sefika, R.H.Campbell y W.S.Liao. "Dynamic Customization in the μChoices Operating System". *Reflection'96*.
[LYI95]	R.Lea, Y.Yokote y J.-I.Itoh. "Adaptive operating system design using reflection". En *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V).* Pp. 95-100. IEEE Computer Society Technical Committee on Operating Systems and Application Environments (TCOS). IEEE Computer Society Press, 1995
[Mad91]	P.Madany. *An Object-Oriented Approach towards a General Model of File Systems.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1991
[Mae87]	P.Maes. "Concepts and Experiments in Computational Reflection". En Proceedigns of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), SIGPLAN Notices, 22(12),

	1987.
[Mae88]	P.Maes. "Issues in Computational Reflection". *Meta-Level Architectures and Reflection*. Pp. 21-35. P.Maes y D.Nardi, Eds. Elsevier Science Publishers B.V. 1988
[MBK+96]	M.McKusick, K.Bostic, M. Karels, J. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1996.
[McA93]	J.McAffer. "The CodA MOP". En *Proceedings of the OOPSLA'93 Reflection Workshop*. 1993.
[McA95a]	J.McAffer. "Meta-Level Programming with CodA". En *Proceedings of the 9th Conference on Object-Oriented Programming, (ECOOP'95)*, LNCS 952, pág 190-214. Springer-Verlag, 1995.
[McA95b]	J.McAffer. "Meta-Level Architecture support for distributed objects". En *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems (IWOOOS'95)*, Lind, Suecia, 1995.
[MCK91]	P.Madany, R.Campbell y P.Kougiouris. "Experiences Building an Object-Oriented System in C++". En *Proceedings of the Technology of Object-Oriented Languages and Systems Conference*, pág 35-42. J.Bezivis y B.Meyer, eds. Prentice-Hall, 1991
[~Mer]	*The Merlin Object-Oriented System*. URL: http://www.lsi.usp.br/~jecel/merlin.html
[Mey93]	B.Meyer, ed. "Concurrent Object-Oriented Programming". *Proceedigns of OOPSLA'93. Special Issue of Communications of the ACM (CACM)*, 36(9). 1993
[MHM+95]	K.Murata, R.N.Horspool, E.G.Manning, Y.Yokote y M.Tokoro. "Unification of Active and Passive Objects in an Object-Oriented Operating System". *Proceedings of 1995 International Workshop on Object Orientation in Operating Systems(IWOOOS'95)*. 1995
[MJD96]	J.Malenfant, M.Jacques y F-N. Demers. "A Tutorial on Behavioral Reflection and its Implementation". En *Proceedings of the Reflection'96 Conference*, 1996.
[MLR+88]	P.Madany, D.Leyens, V.Russo y R.Campbell. " A C++ Class Hierarchy for Building UNIX-Like File Systems". En *Proceedings of the USENIX C++ Conference*, Octubre 1988. También como Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign.
[McMa97]	C.McManis. "Take and in-depth look at the Java Reflection API". *JavaWorld*, Septiembre, 1997.
[MMC95]	P.Mulet, J.Malenfant y P.Cointe. "Towards a Methodology for Explicit Composition of MetaObjects". En *Proceedings of OOPSLA'95*. Pág 316-330. 1995
[MSL+91]	B.D.Marsh, M.L.Scott, T.J.LeBlanc y E.P.Markatos. "First-class user-level threads". En *Proceedings fo the Thirteenth ACM Symposium on Operating*

	System Principles. Pp. 110-121. ACM SIGOPS. 1991.
[MWI+91]	S.Matsuoka, T.Watanabe, Y.Ichisugi y A.Yonezawa. "Object-Oriented Concurrent Reflective Architectures". En *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*. M.Ibrahim, ed. 1991.
[MWY90]	S.Matsuoka, K.Wakita y A.Yonezawa. "Synchronization constraints with inheritance: What is not possible – So what is?". Technical Report 90-010, Department of Information Science, University of Tokyo.
[MY90]	S.Matsuoka y A.Yonezawa. "Metalevel Solution to Inheritance Anomaly in Concurrent Object-Oriented Languages". En *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*. Octubre 1990.
[MY93]	S.Matsuoka y A.Yonezawa. "Inheritance Anomaly in Object-Oriented Concurrent Programming Languages". En *Research Directions in Concurrent Object-Oriented Programming Languages*, G.Agha, P.Wegner y A.Yonezawa, Ed. Pág 107-150. The MIT Press.
[NAJ+76]	➔ **PONER LA REFERENCIA**
[Nie87]	O.Nierstrasz. "Active Objects in Hybrid". *Proceedings of the OOPSLA'87*, ACM SIGPLAN Notices, 22(12):243-253, Diciembre 1987.
[Nie89]	O.Nierstrasz. "A Survey of Object-Oriented Concepts". En *Object-Oriented Concepts, Databases and Applications*, ed. W.Kim y F.Lochovsky, pp. 3-21, ACM Press y Addison-Wesley, 1989.
[Nie91]	O.Nierstrasz. "The Next 700 Concurrent Object-Oriented Languages – Reflections on the future of object-based concurrency". En D.C. Tsichritzis, ed. Object Composition, Pág 165-187. Centre Universitaire d'Informatique. University of Geneva.
[Nie93]	O. Nierstrasz. "Composing Active Objects. The Next 700 Concurrent Object-Oriented Languages". En *Research Directions in Object-Based Concurrency*, Pag. 151-171, ed. G. Agha, P. Wegner, A. Yonezawa. The MIT Press, 1993.
[Nut97]	G.Nutt. *Operating Systems. A Modern Perspective.* Addison-Wesley. 1997
[Obj99]	ObjectSpace. *ObjectSpace Voyager. ORB 3.0 Developer Guide*. ObjectSpace Inc. 1997-1999
[OHE97]	R. Orfali, D. Harkey y J. Edwards. *Instant CORBA*. Wiley Computer Publishing. 1997
[OMG96]	Object Manager Group. *The Common Object Request Broker: Arquitecture and Specificaction*. OMG, 1996
[OMG97]	Object Manager Group. *A Discussion of the Object Management Arquitecture*. OMG, 1997
[Ort97]	F.Ortín Soler. *Diseño y Construcción del Sistema de Persistencia en Oviedo3*. Proyecto Final de Carrera 972001. Escuela Técnica Superior de I. I. e Ingenieros Informáticos. Universidad de Oviedo. Septiembre, 1997.

[Pa97]	M.Papathomas y A.Andersen. "Concurrent Object-Oriented Programming in Python with ATOM". En *Proceedings of the 6th International Python Conference*. 1997
[Pap89]	M.Papathomas. "Concurrency Issues in Object-Oriented Languages". En D. Tsichritzis, ed. *Object-Oriented Development Technical Report*. Pag 207-245. Centre Universitaire d'Informatique, University of Geneva.
[Par76]	D.Parnas. "On the design and development of program families". *IEEE Transactions on Software Engineering*, SE-2(1):1-9, 1976
[PK90]	M.Papathomas y D.Konstantas. "Integrating Concurrency and Object-Oriented Programming – An Evaluation of Hybrid". En D.Tsichritzis, ed. Object Management, Pág 229-244. Centre Universitaire d'Informatique, University of Geneva.
[RAA+88]	M.Rozier, V.Abrossimov, F.Armand, I.Boule, M.Gien, M.Guillemont, F.Herrman, C.Kaiser, S.Langois, P.Leonard y W.Neuhauser. "CHORUS Distributed Operating Systems". *Computing Systems Journal*, 1(4), 1988
[Ras86]	R.Rashid. "From RIG to Accent to Mach: The Evolution of a Network Operating System". *Proceedings of the ACM/IEEE Computer Society Fall Joint Computer Conference*. Pp. 1128-1137. Noviembre, 1986. También como Technical Report, Computer Science Department, Carnegie-Mellon University. 1986
[Rei97]	S.Reitzner. "Splitting Synchronization from Algorithmic Behaviour. An Event Model for Synchronizing Concurrent Classes". En *ECOOP'97, 7th Workshop for PhD Students in Object-Oriented Systems.* E.Ernst, L.Wohlrab y F.Gerhardt, eds. Pag 91-98.
[RF77]	M.Ruschitzka y R.S.Fabry. "A unifying approach to scheduling". *Communications of the ACM*, 20(7):469-476. Julio, 1977
[Rit97]	A.Rito da Silva. "A Quality Design Solution for Object Synchronization". En Proceedings of the European Conference on Parallel Processing. Springer-Verlag, LNCS 1300, Agosto 1997.
[Riv88]	J.des Rivières. "Control-Related Meta-Level Facilities in LISP". *Meta-Level Architectures and Reflection*. Pp. 201-109. P.Maes y D.Nardi, Eds. Elsevier Science Publishers B.V. 1988
[Riv96]	F.Rivard. "Smalltalk: A Reflective Langage". *En Proceedings of the Reflection'96*. G.Kiczales, ed.
[RJC88]	V.Russo, G.Johnston y R.Campbell. "Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques". Technical Report UIUCDCS-R-88-1415, TTR88-4, University of Illinois at Urbana-Champaign, Department of Computer Science, Septiembre 1988.
[RK96]	T.Riechmann y J.Kleinöder. "User-Level Scheduling with Kernel Threads". *Technical Report TR-I4-96-05. University of Erlangen-Nürnberg*.
[RM87]	M.Rozier y J.L.Martins. "The CHORUS distributed operating system: Some design issues". En *Distributed Operating Systems. Theory and*

	Practice. Pp. 262-287. Springer-Verlag
[Rod00]	➔ **PFC Alberto**
[RTY+87]	R.Rashid, A.Tevanian, M.Young, D.Golub, R.Baron, D.Black, W.Bolosky, J.Chew. "Machine Independent Virtual memory Management for Paged Uniprocessor and Multiprocessor Architectures". En *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems.* Pp. 31-39. 1987
[Rum96]	J.Rumbaugh. *OMT Insights: Perspectives on Modeling from the Journal of Object-Oriented Programming*. Cambridge University Press, Octubre 1996
[Rus91a]	V. Russo. "Object-Oriented Operating System Design". IEEE Technical Committee on Operating Systems and Application Environments Newletter, 5(1):34-38, 1991
[Rus91b]	V. Russo. "An Object-Oriented Operating System". PhD Thesis, University of Illinois at Urbana-Champaign, 1991
[RWB97]	R. Raje, J. Williams y M. Boyles. "An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java". *ACM Workshop on Java for Science and Engineering Computation*. En Concurrency: Practice and Experience Journal, Vol. 9(11), 1207-1211. G.Fox, ed. John Wiley & Sons, Ltd., Noviembre 1997.
[Sch93]	F. Schubert. "Objects, Activities and Communication". *Technical Report, TU Chemnitz / International Computer Science Institute (ICSI)*. Berkeley. 1994
[Sch96]	F. Schubert. "Dynamic adaptability in Operating Systems by Means of an Object-Oriented Architecture". *Adaptability in Object-Oriented Software Development Workshop. 10th European Conference on Object-Oriented Programming, (ECOOP'96)*. 1996
[SG98]	A.Silberschatz, P.Galvin. *Operating System Concepts.* 5th edition. Addison-Wesley Publishing Company. 1998
[SGH+89]	M.Shapiro, Y.Gourhant, S.Habert, L.Mosseri, M.Ruffin y C.Valot. "SOS: An Object-Oriented Operating System – Assessment and Perspectives". *Computing Systems*, 2(4), 1989
[Sha91a]	M. Shapiro. "Object-Support Operating Systems". En Proceedings of the *Workshop on Operating Systems on Object Orientation. ECOOP-OOPSLA*. Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments, 5(1):39-42, 1991
[Sha91b]	M. Shapiro. "Soul: an object-oriented OS framework for object support". En Proceedings of *The Workshop on Operating Systems for the Nineties and Beyond*. Springer-Verlag. 1991
[Smi82]	B.C.Smith. "Reflection and Semantics in a procedural language". PhD Thesis. *Technical Report 272 Massachusetts Institute of Technology. Laboratory for Computer Science*. Cambridge, Massachusetts, 1982.
[Smi84]	B.C.Smith. "Reflection and Semantics in Lisp". En *Proceedings of*

	Principles of Programming Languages. 1984.
[Smi90]	B.C.Smith. "What do you mean, Meta?". En Proceedings of the *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'90).* ACM Press, 1990.
[SPr93a]	W.Schröder-Preikschat. "Object-Orientation in a Family of Parallel Operating Systems". En *Software Technology*, volumen 2 de *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, T.Lewis Hesham El-Rewini y B.Shriver, eds. Pág 60-69. IEEE Computer Society Press, 1993
[SPr93b]	W.Schröder-Preikschat. "Design Principles of Parallel Operating Systems". *International Summer Institute of Parallel Computer Architectures, Languages and Algorithms.* 1993
[SPr94]	W.Schröder-Preikschat. *The Logical Design of Parallel Operating System*. Prentice-Hall, 1994
[SPr95]	W.Schröder-Preikschat. "Experience Developing an Object-Oriented Parallel Operating System". IBM Symposiun über Objektorientierte Betriebssystems, 1995
[SS95]	C.Small y M.Seltzer. "Structuring the Kernel as a Toolkit of Extensible, Reusable Components". En *Proceedings of the Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS'95).* Pp. 134-137. IEEE Computer Society, IEEE Computer Society Press, 1995
[Sta98]	W.Stallings. *Operating Systems. Internals and Design Principles.* 3rd edition. Prentice Hall. 1998
[Str92]	B.Stroustrup. "Run Time Type Identification for C++". En *USENIX C++ Conference Proceedings*, Portland, 1992.
[Str93]	B. Stroustrup. *El Lenguaje de Programación C++*, 2ª Edición. Addison-Wesley/Díaz de Santos. 1993
[Sun97a]	Sun Microsystems. *Java Virtual Machine Specification*. 1997.
[Sun97b]	Sun Microsystems. *Java Core Reflection, API and Specification*. Febrero, 1997.
[Tan94]	A.Tanenbaum. *Sistemas Operativos Modernos*. Prentice-Hall. 1994
[TD99]	L.Tajes Martínez y M.A.Díaz Fondón. "System Object Model (SOM)". En *Handbook of Object Technology*, pág 30-1 a 30-15. Saba Zamir, ed. CRC Press. 1999.
[Tho89]	D. Thomas. "What's in an Object?". *BYTE*. March, 1999. Pág. 231-240
[TNO92]	H.Tokuda, T.Nakajima y S.Oikawa. "Towards a New Operating System Architecture: Microkernel vs. Reflective Architecture". En *10th Conference Proceedings, Japan Society for Software Science and Technology*. 1992
[TRS+90]	A.S.Tanenbaum, R.van Renesse, H.van Staveren, G.J.Sharp, S.J.Mullender, J.Jansen y G.van Rossum. "Experiences with the Amoeba Distributed

	Operating System". *Communications of the ACM*, 33(12), 1990.
[TW98]	A.S.Tanenbaum y A.S. Woodhull. *Sistemas Operativos. Diseño e Implementación*. 2ª edición. Prentice Hall. 1998
[VA98]	C.A. Varela y Gul A. Agha. "What after Java? From objects to actors". En *Proceedings of the Seventh International World-Wide Web Conference*. Brisbane, Australia, 1998.
[Vah96]	U.Vahalia. *Unix Internals. The New Frontiers*. Prentice-Hall, 1996
[Var94]	P. Varhol. "Los núcleos pequeños lo hacen mejor". *Binary*, Febrero 1994
[VBJ+96]	J.Van Oeyen, S.Bijnens, W.Joosen, B.Robben, F.Matthijs y P.Verbaeten. "A Flexible Object Support System as Run-time for Concurrent Object-Oriented Languages". En *Metaobject Protocols*, Capítulo 12, C.Zimmermann, editor. CRC Press, Mayo 1996
[Ven98]	B.Venners. *Inside the Java Virtual Machine*. McGraw-Hill. 1998.
[Weg90]	P. Wegner. "Concepts and Paradigms of Object-Oriented Programming". *OOPS Messenger*. 1(1). Agosto 1990. Pág. 7-87
[WF88]	M.Wand y D.P.Friedman. "The Mistery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower". En *Meta-Level Architectures and Reflection*. P.Maes y D.Nardi, eds. Pag. 111-133. Elsevier Science Publishers. 1988
[Wir90]	N.Wirth. *Programming in Modula-2. 4th edition*. Springer-Verlag. 1990
[WS95]	Z.Wu y R.J.Stroud. "Using Metaobject Protocols to Structure Operating Systems". *Proceedings of the Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS95).* Pp. 228-231. IEEE Computer Society, IEEE Computer Society Press, 1995
[WY88]	T.Watanabe y A.Yonezawa. "Reflection in an Object-Oriented Concurrent Language". En *Proceedigns of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'88).* SIGPLAN Notices, Vol 23, Pág 306-315. ACM Press, 1988.
[YMF+91]	Y.Yokote, A.Mitsuzawa, N.Fujinami y M.Tokoro. "Reflective Object Management in the Muse Operating System". IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS'91).
[Yok92]	Y.Yokote. "The Apertos Reflective Operating System: The Concept and Its Implementation". En Proceedings of the *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92).* Pp. 414-434. A.Paepcke, ed. ACM Special Interest Group on Programming Languages, ACM Press, October 1992. Also SIGPLAN Notices 27(10), October 1992
[Yon90]	A.Yonezawa, ed. *ABCL. An Object-Oriented Concurrent System*. Computer Systems Series. The MIT Press. 1990.
[YSH+90]	A.Yonezawa, E.Shibayama, Y.Honda, T.Takada y J.-P.Briot. "An Object-Oriented Concurrent Computation Model ABCM/1 and its Description Language ABCL/1". En *[Yon90]*.

[YST+87]	A.Yonezawa, E.Shibayama, T.Takada y Y.Honda. "Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1". En *Object-Oriented Concurrent Programming*, A.Yonezawa y M.Tokoro, eds. Pág 55-89. The MIT Press, Cambridge, Massachussets, 1987.
[YT87a]	Y.Yokote y M.Tokoro. "Concurrent Programming in ConcurrentSmalltalk". En [YT87b]. Pág 129-158. The MIT Press, 1987.
[YT87b]	Y.Yokote y M.Tokoro (Eds.). *Object-Oriented Concurrent Programming*. The MIT Press, 1987.
[YT88a]	A.Yonezawa y M.Tokoro. "Object-Oriented Concurrent Programming: An Introduction". En *[YT88b]*.
[YT88b]	A.Yonezawa y M.Tokoro, eds. *Object-Oriented Concurrent Programming*. Computer System Series. The MIT Press, 1988.
[YTM+91]	Y.Yokote, F.Teraoka, A.Mitsuzawa, N.Fujinami y M.Tokoro. "The Muse Object Architecture: A New Operating System Structuring Concept". En *ACM Operating Systems Review*, 25(2):22-46, Abril, 1991. También como: SCSL-TR-91-002 Sony Computer Science Laboratoy Inc.
[YTR+87]	M.Young, A.Tevanian, R.Rashid, D.Golub, J.Eppinger, J.Chew, W.Bolosky, D.Black y R.Baron. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System". En *Proceedings of the 11th Symposium on Operating System Principles*. Pp. 63-76. ACM Special Interest Group on Operating Systems, Noviembre 1987.
[YTT89]	Y.Yokote, F.Teraoka y M.Tokoro. "A Reflective Architecture for an Object-Oriented Distributed Operating System". En S. Cook, ed. Proceedings of *European Conference on Object-Oriented Programming(ECOOP'89)*,89-106. Cambridge University Press, 1989.
[YTY+89]	Y.Yokote, F.Teraoka, M.Yamada, H.Tezuka y M.Tokoro. "The Design and Implementation of the Muse Object-Oriented Distributed Operating System". En *Proceedings of 1st Conference on Technology of Object-Oriented Languages and Systems*, 1989. También como SCSL-TR-89-010 Sony Computer Science Laboratory Inc.
[YW89]	A.Yonezawa y T.Watanabe. "An introduction to object-based reflective concurrent computation". En *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming. SIGPLAN Notices*, 24:50-54. ACM Press, Abril, 1989.
[ZC96]	C.Zimmermann y V.Cahill. "It's your choice – On the design and Implementation of a flexible Metalevel Archictecture". En *Proceedings of the 3rd International Conference on Configurable Distributed Systems.* IEEE Computer Society Press, Mayo 1996.