Departamento de Informática

Universidad de Oviedo

TESIS DOCTORAL

CLASIFICACIÓN DE USUARIOS BASADA EN LA DETECCIÓN DE ERRORES USANDO TÉCNICAS DE PROCESADORES DE LENGUAJE

Presentada por:
Juan Ramón Pérez Pérez
Para la obtención del título de Doctor por la Universidad de Oviedo

Dirigida por el
Profesor Doctor D. Juan Manuel Cueva Lovelle

Oviedo, enero de 2006

Resumen

Esta tesis lleva a cabo la definición de un modelo para un entorno de desarrollo de software, que facilita a los usuarios la construcción de aplicaciones con una mejor calidad del código que los entornos actuales.

El modelo que, está implementado en el sistema denominado SICODE, se basa en técnicas de procesadores de lenguaje que permiten realizar un análisis estático del código fuente, para descubrir los errores de programación presentes en el código. Este modelo pretende incidir sobre los estilos en la forma de programar y no sólo sobre errores puntuales. Para ello se ha establecido lo que se denomina una *historia de compilación*, sobre la que se generan métricas de evolución y frecuencia de errores a lo largo del tiempo. Se utiliza un modelo activo en la prevención de errores mediante el envío de avisos. Estos avisos permiten asociar información semántica a cada error en una base de conocimientos. Esta base de conocimientos es dinámica y se realimenta con la experiencia de todos los desarrolladores, lo que permite aprender de la experiencia de los demás.

En esta tesis se ha realizado un estudio que permite llevar a cabo la clasificación de usuarios basándose en la detección de errores en el código. SICODE nos ha permitido analizar los errores de un amplio número de proyectos realizados por desarrolladores. Así, hemos caracterizado distintos tipos de errores dependiendo de la experiencia de los programadores. Este estudio proporciona datos que permitirán construir las bases para modelizar al usuario y de esta forma hacer que el modelo sea adaptable.

La tesis ofrece nuevos enfoques sobre los sistemas de desarrollo existentes en la actualidad. La construcción de la *historia de compilación* permite un análisis más profundo que el que puede realizar un simple compilador, y este análisis está al servicio de la mejora del código. Además, el sistema está centrado en el entorno de desarrollo y fusiona el propio desarrollo de software con el aprendizaje para hacer un código de calidad: el usuario simplemente programa y SICODE le proporciona medios para mejorar el estilo de programación. Por último, la colaboración se utiliza, no sólo para facilitar el desarrollo, sino también en la construcción de una base de conocimientos, lo cual permite construir un sistema que se realimente con el intercambio de experiencias entre los usuarios.

Palabras clave

Entorno de desarrollo integrado, análisis estático de errores, base de conocimientos, historia de trabajo, historia de compilación, CSCL (Computer Supported Cooperative Learning), calidad del código fuente, unidad de compilación, aprendizaje basado en errores.

Abstract

This thesis defines a model for a development environment of software that makes easier the construction of the applications with an improvement of the code quality that the current environments.

The model, we have implemented in the system called SICODE, is based on language processor techniques which make it possible to carry out a static analysis of the source code. With this model we want to see is styles of programming and not just isolated errors. For this purpose the *compilation history* is used that lets us produce statistics of error evolution and frequency during a period of time. An active model is used in the prevention of errors by means of sending of warnings. These warnings allows to link semantic data to each error in the knowledge base. This knowledge base is dynamic and it is feedback with the experience shared by all developers, this permits to learn of the experience of other.

A research that allows classifies the users based in code errors, has been carried out in this thesis. The SICODE system has been use to analyse a wide number of projects wrote to different developers. This way we have characterized different types of errors according to the experience of the developer. This investigation provides data which will make it possible to build up the foundation of a user model and so the system will be adapted to each user.

This thesis makes new approaches to the nowadays existing development systems. The building of the compilation history allows a deeper analysis than that of a simple compiler and it helps to improve the code. Besides, the system focuses on the development environment and merges development of software with learning for developing an improvement code: the user only has to make a program and SICODE provides means of improvement the programming style. Finally, the collaboration is used not just in the development, but also in the construction of a knowledge base providing the exchange of the experiences between the users.

Keywords

Integrated development environment, static analysis of errors, knowledge base, work history, compilation history, CSCL (Computer Supported Cooperative Learning), source code quality, compilation unit, learning based on errors.

Agradecimientos

Quisiera reflejar en estas líneas mi agradecimiento a todos mis compañeros, muy especialmente a los integrantes del grupo del Laboratorio de Tecnologías Orientadas a Objetos (OOTLab), por todo el apoyo tanto en la parte científica como en la parte personal que me han proporcionado durante la elaboración de esta tesis doctoral, y que de una forma u otra han colaborado a que pudiera llegar hasta aquí.

A Cueva, mi director de tesis, que con sus sabios consejos me ha guiado en la escritura de esta tesis haciendo posible que este documento tuviera forma y porque estuvo siempre ahí, apoyándome y caminando a mi lado cuando más lo necesité.

A mis compañeros del café, Macamen, Marián, Fernando y Lourdes, por permitirme disfrutar junto a vosotros de esos pequeños momentos de esparcimiento y porque las ideas muchas veces surgen a partir de conversaciones de café.

A todos los estudiantes que han tenido que ver con SICODE, especialmente a Ramón, Daniel, Xuan y Cristina, por las ideas que han aportado y porque en las conversaciones con ellos el sistema ha crecido y se ha perfeccionado.

A mis padres porque gracias a ellos soy lo que soy. A mis hermanos, Joaquín y Pablo, porque siempre confiaron en mí y son un ejemplo de cómo hacer bien el trabajo.

A mi mujer, Puerto, por tener paciencia conmigo, por darme fuerzas cuando las mías flaqueaban, por escuchar cuando le planteaba cosas sobre la investigación y aportar un montón de cosas, en definitiva por darme tanto amor. A mi hija Laura porque me ha cambiado la forma de ver la vida y eso me ha proporcionado un nuevo espíritu para afrontar retos como este.

Juan Ramón, enero de 2006

Tabla de Contenidos

Capítulo 1. Introducción

En este capítulo hacemos un planteamiento general del problema que abordamos en la tesis y los objetivos globales que pretendemos conseguir. En el último apartado, describimos la organización de todo el documento.

1.1 Planteamiento del problema

La aparición de errores es algo común en el proceso de desarrollo del software. Los errores siempre conllevan pérdidas de tiempo y dinero en el proceso de desarrollo, en ocasiones, los errores llegan a la fase en la que producto software debe ser utilizado por los usuarios finales, lo que implica productos de baja calidad que causan defectos tanto funcionales como no funcionales. Esta baja calidad del software supone un grave problema. Algunos informes dan datos como que *el 60% de los desarrolladores de software en Estados Unidos están involucrados en resolver errores que se podrían haber evitado* [Jones, 1998].

Estos problemas pueden reducirse utilizando distintas técnicas de ingeniería del software que dan buenos resultados y se hacen imprescindibles para obtener software de calidad [Pressman, 1997]. Sin embargo, dentro del proceso general de construcción de software, es básico fijar la atención en el subproceso de escritura de código de la aplicación. En este subproceso, la interacción entre el programador y el entorno de desarrollo es fundamental para lograr reducir la cantidad de defectos introducidos. Actualmente, los compiladores y otras herramientas de análisis de código permiten la detección automática de distintos tipos de errores. Utilizando esta información, el desarrollador puede eliminar estos errores a través de un proceso de depuración. Esto es fundamental para obtener un producto software de alta calidad; según Allen [Allen, 2002] *existe una interdependencia crucial entre la depuración efectiva y el desarrollo efectivo.*

Para conseguir una alta calidad del software, es muy importante tener programadores con experiencia ya que la calidad y seguridad del software producido depende inevitablemente de la destreza y experiencia de los programadores involucrados. Además, según Allen: *hay pocos programadores experimentados para la demanda que existe. Para formar a más programadores es insuficiente la enseñanza tradicional de conocimiento teórico. Necesitamos transmitir habilidades prácticas en el desarrollo de sistemas robustos, esto lleva años de experiencia* [Allen, 2002]. Parte de este conocimiento práctico consiste en la habilidad de diagnosticar eficientemente y subsanar los errores del sistema software. La depuración efectiva está lejos de ser una habilidad trivial. Buscar y eliminar errores ocupa una porción significativa del tiempo de desarrollo en un proyecto software. Si esta tarea se pudiera realizar de forma más eficiente, el software resultante podría ser más fiable y el proceso de desarrollo sería más rápido. Por tanto, es fundamental el aprendizaje de **técnicas de detección y corrección de errores** que sirvan para mejorar el software ya creado y se puedan utilizar en la mejora del proceso general de codificación. El aprendizaje de estas técnicas no es sencillo y para alcanzar un dominio

adecuado es necesario complementar los conocimientos teóricos con la experiencia práctica que ayude a la asimilación de estos conceptos. Además, es importante que lo aprendido en no sólo sirva para la depuración de errores sino poder aplicarlo a la codificación para evitar que se produzcan nuevos errores y poder hacer más eficiente todo el ciclo de desarrollo.

La adquisición de habilidades y experiencia en el ciclo de desarrollo que debe realizar un desarrollador no se limita a una etapa de formación en el inicio de la vida. Los desarrolladores deben de estar aprendiendo continuamente porque todos los días surgen nuevas tecnologías y tendencias mientras que otras se quedan obsoletas. Según Ivar Jacobson [Jacobson 2002]: *El desarrollo de software nunca ha sido tan complejo como lo es ahora. Los desarrolladores de software trabajan intensivamente con el conocimiento. No sólo deben comprender las nuevas tendencias y tecnologías, necesitan saber como aplicarlas de forma rápida y productiva.*

Pese a la gran cantidad de trabajos y los múltiples enfoques existentes en el campo de la calidad en el desarrollo de software, hay muy pocas propuesta relacionadas con la detección, análisis y corrección de errores en el código del programa [Luján-Mora, 2003]. En la práctica, esta es una faceta que cada desarrollador va aprendiendo, en la mayoría de los casos de forma autodidacta, por medio de la experiencia que va adquiriendo utilizando el ensayo y error en el desarrollo de proyectos. Los libros de lenguajes de programación prácticamente no tratan el tema y los manuales se limitan a proporcionar una serie de instrucciones de manejo de las herramientas; pero no profundizan en la aplicación de las técnicas. Los enfoques científicos plantean modelos de difícil aplicación práctica [Hovemeyer, 2004b]. Las únicas propuestas a las que se pueden acoger los desarrolladores son libros que describen una serie de reglas prácticas para mejorar el desarrollo de aplicaciones.

Los compiladores automatizan la comprobación de alguna de estas reglas prácticas y tras un proceso de compilación emiten mensajes indicando su violación. Sin embargo, los mensajes de error de los compiladores no facilitan la labor de la corrección en la aplicación de las reglas, ya que muchas veces son crípticos y ambiguos. Esto causa que sean difíciles de entender para un desarrollador con poca experiencia. Además de ser difíciles de comprender, *los compiladores solamente proporcionan síntomas de defectos y es necesario entender dónde y cual es el problema* [Humphrey, 1997]. Muchas veces, pese a disponer de la información que proporciona el compilador no es sencillo relacionar los síntomas indicados en los mensajes de error con los problemas reales para eliminarlos y tratar de evitarlos en el futuro. Según Humphrey [Humphrey, 1997]: *Es importante separar la cuestión de encontrar o identificar los defectos de la determinación de sus causas. El primer paso para gestionar los defectos es entenderlos. Para hacer eso se deben reunir los datos de los defectos. Entonces, se podrán entender esos errores y comprender cómo evitarlos. También se podrá comprender como encontrarlos mejor, corregirlos o prevenir los defectos que todavía se introducen.*

Por lo tanto, es necesario la creación y uso de **herramientas que no sólo detecten errores sino que contabilicen, hagan un seguimiento y permitan un análisis y clasificación** de estos errores. Además se debe añadir la suficiente **información semántica a los errores, de forma que permita al desarollador la interpretación del mensaje de error en su contexto, la relación con sus causas y las posibles soluciones que permitan eliminar el error**. Por último, los entornos no deben de ser pasivas y esperar a que el usuario pida información sino que deben **indicar al usuario los errores en el momento que se produzcan mediante avisos activos**.

Un punto básico que ayudaría a que todo el proceso se desarrollase de forma eficiente es que los desarrolladores trabajasen en forma colaborativa. Hay muchos sistemas colaborativos que se han aplicado con éxito tanto a situaciones de desarrollo de software [Shen, 2000] (CSCW, Computer Supported Cooperative Work), como en situaciones de

aprendizaje [Bravo et al, 2004] (CSCL, Computer Supported Cooperative Learning) Consideramos que es fundamental que cualquier entorno de desarrollo proporcione facilidades para desarrollar el trabajo en grupos de forma adecuada, y proporcione herramientas tanto para compartir archivos como para coordinarse en todas las tareas involucradas en el desarrollo. Otro ámbito en el que la colaboración va a aportar un valor añadido es la construcción de una base de conocimientos que, a través de la experiencia de los desarrolladores permitirá añadir información semántica a los distintos mensajes de error recogidos por el sistema.

Por último, no queremos que la utilidad del sistema se limite a la corrección de errores puntuales, debería permitir elaborar una guía adaptada al perfil de cada desarrollador extraído del análisis de errores. De esta forma se proporcionaría una herramienta para mejorar el estilo de programación y evitar futuros errores.

En la tesis se plantean dos contextos de aplicación para la herramienta descrita: un contexto académico donde los estudiantes utilizan el sistema en grupos para aprender técnicas de programación, construcción de software y lenguajes de programación y un contexto profesional donde los desarrolladores tienen que adquirir el dominio de nuevas técnicas y producir software de alta calidad.

1.2 Objetivos de la tesis

En este apartado se enuncian los objetivos que se pretenden desarrollar en esta tesis visto el planteamiento previo. Existen dos grandes ejes que marcan la línea de investigación: el modelado del sistema y la utilización de éste para el análisis de datos orientado a la clasificación de los desarrolladores; cada uno de estos ejes conlleva la consecución de varios sub-objetivos. A continuación, se describen dichos ejes.

1.2.1 Modelado de un sistema que permita la mejora de la calidad del código fuente

El primer objetivo de esta tesis es el modelado de un sistema que permita superar los inconvenientes de los sistemas actuales en la construcción de software de calidad descritos, muy brevemente, en el apartado anterior.

El sistema esta fundamentado en las siguientes bases:

1. **Sistema orientado a la mejora de la calidad del código fuente.** Permite detectar, eliminar y prevenir errores de forma más eficiente a través técnicas de procesadores de lenguaje. Para ello dispone de las siguientes características:
   - Captura y almacenamiento on-line de errores y *warnings* mediante técnicas de procesadores de lenguajes. Se recogen en tiempo real los problemas existentes en el código mientras los desarrolladores están escribiendo.
   - Mantenimiento de una *historia de errores*, facilitando el seguimiento y la monitorización de usuarios individuales y de grupos de trabajo en relación con los errores de programación.
   - Análisis de los errores de programación cometidos por los desarrolladores. Para ello se estudian distintas métricas para aplicarlas sobre los errores individuales y de grupo recogidos en la base de datos, con el fin de definir indicadores personalizados para cada desarrollador.

2. **Entorno que proporciona soporte para equipos virtuales de desarrollo y permita aprender de los errores propios**. El sistema debe proporcionar distintos servicios a los desarrolladores a través de un entorno de desarrollo.
   - Los desarrolladores deben disponer de un Entorno Integrado de Desarrollo sobre Internet.
   - Los desarrolladores pueden aprender y adquirir experiencia en las nuevas técnicas de construcción de aplicaciones mediante el desarrollo de proyectos software aprendiendo así de los errores propios.
   - Los desarrolladores pueden colaborar en el desarrollo y revisión de programas. Se pretende facilitar el funcionamiento de equipos virtuales de desarrollo mediante herramientas de comunicación y colaboración. Todos los desarrolladores tendrán acceso al código de los demás, permitiendo la realización de revisiones y mejoras sobre el código fuente.
   - Los desarrolladores disponen de información activa y adaptada que les ayuda a realizar correctamente la escritura del código. Se pretende que el entorno, mediante la emisión de avisos, proporcione información relevante y adaptada a los desarrolladores mientras escriben el código.
3. **Base de conocimientos colaborativa** que asocia información semántica a cada uno de los tipos de errores. A través de esta base de conocimientos, los usuarios pueden consultar la información relacionada con los errores surgidos en el análisis de su código.
   - El sistema permite una asociación automática de los distintos mensajes de error con la información disponible en la base de conocimientos.
   - La base de conocimientos dispondrá de información contextualizada y derivada de la experiencia de los desarrolladores lo que permite que cualquier desarrollador pueda aprender de la experiencia de los demás.
   - La base de conocimientos es dinámica. Permite que la información contenida crezca y el sistema se realimente con la información que los desarrolladores van generando.

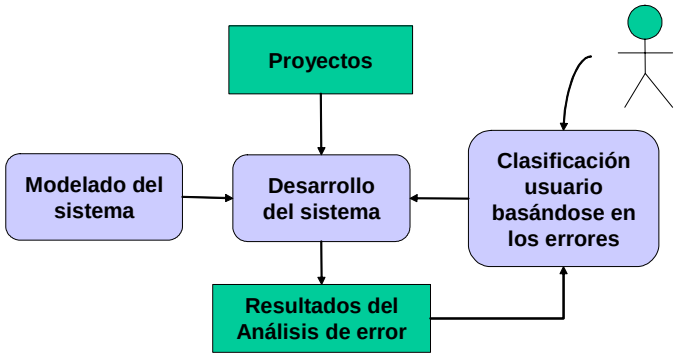

Figura 1. Representación esquemática de los objetivos de la tesis

1.2.2 Clasificación de usuarios basada en la detección de errores

El segundo gran objetivo es la utilización del sistema modelado en el objetivo anterior para recoger y analizar datos que permitan clasificar a los usuarios del sistema (desarrolladores) utilizando las estadísticas de errores. La consecución de este objetivo conlleva:

1. **Caracterizar los errores** que cometen los desarrolladores en distintos niveles de aprendizaje basándose en el sistema con los objetivos descritos en el apartado anterior. Para ello se analizarán los errores de una amplia muestra de proyectos de distintos niveles académicos.
2. **Obtener un perfil de desarrollador** dependiendo de su experiencia, utilizando los datos del análisis anterior.
3. **Desarrollar una guía adaptada al perfil de cada usuario** que permitirá al propio desarrollador conocer mejor los errores cometidos a la hora de programar para evitarlos y así, construir aplicaciones de mayor calidad.

La información extraída en este objetivo se podrá utilizar para realimentar la base de conocimientos del sistema y así proporcionará permitirá una adaptación al perfil del usuario.

1.3 Organización del documento

A continuación describimos la organización de este documento por capítulos:

Introducción y problema a resolver. En el capítulo 1 se hace una introducción general planteando el problema abordado y los objetivos generales que queremos alcanzar con esta Tesis. En el capítulo 2 se describen con mayor profundidad los retos de un sistema que pretenda hacer más fácil la utilización de lenguajes de programación, obteniendo un código fuente de calidad: sin errores y fácilmente mantenible.

Estado del arte. En el capítulo 3 se estudian de forma crítica otros sistemas que desde distintas perspectivas intentan abordar el problema planteado y se identifican sus puntos fuertes y sus carencias.

Modelo del sistema. En el capítulo 4 se plantean los requisitos del modelo para un sistema que permita dar solución al problema planteado.

Implementación del sistema. En los capítulos siguientes se describen y analizan diferentes prototipos desarrollados del sistema definido en el capítulo 4:

- Sistema SICODE. Capítulo 5. Descripción global del sistema y explicación de su división en subsistemas y de las decisiones comunes que se han tomado en su implementación.
- Compilador Web SICODE. Capítulo 6. Describe el primer prototipo desarrollado en el que se diseñan las diferentes características del sistema para someterlas a una primera evaluación.
- Sistema de análisis de errores de programas: PBA. Capítulo 7. Se plantea un prototipo para el subsistema de captura, almacenamiento, análisis de errores y generación de informes sobre los mismos.
- Sistema para la colaboración en el desarrollo de aplicaciones: COLLDEV. Capítulo 8. Se explica el prototipo que desarrolla el subsistema que permite la comunicación entre los distintos desarrolladores de un proyecto, la colaboración en el desarrollo

de los distintos usuarios utilizando un almacén común para los proyectos software y la coordinación en las tareas de desarrollo y corrección de errores.

- Entorno de desarrollo integrado en Web y base de conocimientos colaborativa: IDEWeb. Capítulo 9. Se describe el subsistema de entorno de Web de desarrollo y la base de conocimientos que permite obtener a los usuarios información sobre las causas de un error y como corregirlo.

Experimento de clasificación de usuarios. En el capítulo 10 se realiza un estudio del comportamiento de los desarrolladores en distintos niveles de aprendizaje. Recopilando un número relevante de proyectos software que se analizan con nuestro sistema. Así podemos comparar los tipos de errores que cometen los usuarios dependiendo de lo expertos que sean en las técnicas de programación.

Conclusiones y trabajo futuro. En el capítulo 11 se describen las aportaciones de este trabajo y las conclusiones extraídas y las posibles líneas de investigación que se abren a partir de ahora.

Apéndices. Se incluyen cuatro apéndices con información complementaria:

- Apéndice A. Archivos de configuración del sistema de análisis de errores en programas. Incluye los archivos de configuración XML de distintos aspectos de este subsistema.
- Apéndice B. Información sobre los errores frecuentes. Incluye información sobre los errores más frecuentes analizados en el capítulo 10.
- Apéndice C. Avisos por proyecto e informes de análisis. Se incluyen todos los tipos de errores para cada conjunto de proyectos con su frecuencia. Además incluye los informes generados directamente por el subsistema de análisis de errores en programas.
- Apéndice D. Referencias bibliográficas.

Capítulo 2. Problemática en la calidad del código fuente y en el aprendizaje de la programación

Antes de proceder a explicar los sistemas actuales orientados al aprendizaje y mejora de la calidad del código, es necesario, enmarcar en el contexto adecuado la problemática que tiene un programador principiante; pero también uno experimentado en el aprendizaje de la programación y la mejora de su código fuente. Este capítulo servirá para enfocar y mostrar las dificultades que existen en dicho aprendizaje.

2.1 Dificultades de los programadores principiantes para aprender los conceptos de programación

Satratzemi y sus colegas [Satratzemi, 2001] a partir de [Brusilovski, 1998] [Boulay, 1989] resumen las dificultades que encuentran los programadores a la hora de enfrentarse a un programa:

- Dificultades que aparecen por tener un modelo conceptual de la máquina que difiere del modelo real.
- Dificultades atribuidas a la sintaxis y semántica de los lenguajes de programación.
- La necesidad de comprender las estructuras de programación establecidas.
- La necesidad de aprender cómo diseñar, desarrollar, verificar y depurar un programa con ciertas herramientas.
- El hecho de que los editores, compiladores y depuradores se diseñan por y para programadores profesionales. De tal forma, que tienen una complejidad extra para los programadores principiantes.
- Los entornos de programación no son capaces de ofrecer visualización de la ejecución del programa. Así, los detalles de la ejecución permanecen ocultos y los programadores pueden percibir un programa como una caja negra difícil de comprender. Esta carencia de realimentación visual hace difícil la compresión de la semántica del lenguaje.

Kölling [Kölling, 2003] se plantea la causa de las dificultades en el aprendizaje de la programación y en concreto de la programación orientada a objetos y considera que la programación a objetos no es más compleja que la estructurada; pero la carencia de herramientas apropiadas la hacen más complicada. Kölling [Kölling 1999a] plantea tres

problemas clave de los entornos de desarrollo actuales para la enseñanza de la programación orientada a objetos:

- Los entornos no son orientado a objetos. Un entorno para un lenguaje orientado a objetos no tiene por que ser un entorno orientado a objetos. Sin embargo, el entorno debería reflejar el paradigma del lenguaje. En la mayoría de los entornos hay que trabajar con el sistema de archivos y la estructura de directorios. Además, la interacción del usuario es exclusivamente con líneas de código fuente, no con objetos. La interacción con objetos no está soportada por muchos entornos pese a ser uno de los conceptos fundamentales.
- Los entornos son demasiado complejos. Un ejemplo es la utilización, para programar en Java, del Java SDK de Sun en línea de comandos, lo que provoca los problemas típicos de la línea de comandos. En otros casos se utilizan entornos que están pensados para programadores profesionales y que suponen una sobrecarga en su interfaz para los principiantes. Otros son modificaciones de entornos para lenguajes procedurales que no disponen de las herramientas apropiadas para las abstracciones de la orientación a objetos.
- Los entornos centran al usuario en la construcción de interfaces. Muchos entornos utilizan gráficos; pero para tareas no relevantes para que los estudiantes puedan adquirir los conceptos de orientación a objetos adecuados. En concreto, muchos entornos se concentran en la construcción de interfaces gráficos (GUIs) y esto puede distorsionar la visión de la programación orientada a objetos. Una de las aplicaciones más interesantes de los gráficos sería mostrar la estructura de clases y pocos entornos la utilizan.

2.2 Entornos de programación y sus limitaciones en para el aprendizaje

Los entornos actuales de desarrollo de aplicaciones software aportan de forma integrada un gran número de herramientas: editores, compiladores, depuradores, gestores de configuración, gestores de versiones, etc. Sin embargo, desde el punto de vista de un programador principiante esto, lejos de facilitar su trabajo con el entorno y que este permita un aprendizaje de la programación, crea barreras, a veces, difícilmente superables. Estos entornos no proporcionan un modelo mental coherente con los conceptos básicos de programación y tampoco ayuda a los programadores a construir nuevos conocimientos de programación a partir de conocimientos previos.

Un punto especialmente destacable por la complicación que representa para un programador principiante es la explicación de los errores: normalmente consiste en un mensaje corto, sin más explicación, acompañado de la línea donde se detecta este tipo de error. Esto es suficiente para los programadores profesionales con experiencia, consiste en un simple enlace mental a un tipo de error ya solucionado muchas veces y para el cual se dispone de patrones de solución. Sin embargo, esto es muy críptico para los programadores sin experiencia, por lo que muchas veces tienen que emplear bastante tiempo buscando información relacionada que les indique cual es el problema real indicado por el mensaje relacionándolo, si es necesario, con los conceptos que ya tiene el programador y cuales son los posibles patrones para solucionar el problema.

Por último, los entornos tampoco proporcionan facilidades para integrarlos en la dinámica del aprendizaje. Por una parte son sistemas pasivos que analizan el código de forma superficial (lo imprescindible para poder generar código objeto) y sólo cuando el

desarrollador realiza la compilación [Jacobson 2002]; por otra parte carecen de herramientas que permitan guiar al programador para mejorar su estilo de programación, el único objetivo es corregir errores puntuales en el código.

2.3 La práctica de la programación como medio para aprender y mejorar

Los desarrolladores tienen la máxima de "a programar se aprende programando", es decir, las destrezas necesarias para ser un buen programador se adquieren sobre todo en la práctica, implementando programas, encontrando problemas y buscando soluciones a estos problemas. Sin embargo, los sistemas de aprendizaje de la programación raramente permiten ponerse a programar directamente; se trata más bien de la transmisión de conocimientos sobre programación de forma teórica. Sería necesario plantear un sistema en el que el estudiante pudiese ponerse a programar desde el primer momento incluso antes de saber instalar el compilador y que fuese el propio entorno el que le fuese guiando en su aprendizaje.

2.4 Programar en grupo, colaborando en la solución de errores y en mejora del código fuente

Los entornos de programación están pensados para trabajar individualmente con ellos; normalmente cuando se trabaja en equipo este se encuentra próximo físicamente y los desarrolladores se pueden comunicar directamente sin necesidad de soporte informático. Normalmente el soporte para el trabajo en equipo de desarrollo consiste en un sistema de control de versiones (CVS) [Cederqvist 1993] común que permita compartir sin problemas los archivos del proyecto entre todos los miembros del equipo. Sin embargo, este sistema en si mismo no ayuda a que los programadores se coordinen y colaboren en la mejora de la calidad del código en general y en la solución de errores del código fuente en particular. Es necesario un sistema que además de compartir los ficheros permita la comunicación entre desarrolladores y la toma de decisiones en el proyecto, dando soporte a lo que se denominan equipos virtuales de desarrollo [Scotts 2003].

2.5 Programar a distancia

Es útil, sobre todo a la hora de dar los primeros pasos en el aprendizaje de la programación, el poder programar sin necesidad de tener instalado un compilador, ni un entorno en la máquina local. Si disponemos de un entorno Web que permita realizar una edición de los archivos fuente, una gestión de los archivos del proyecto almacenados en el servidor y una compilación de estos utilizando un compilador del servidor el desarrollador podría ponerse a programar en cualquier ordenador con conexión a Internet sólo con tener instalado un navegador, independientemente del resto del software de que disponga.

2.6 Tutor permanente: qué estoy haciendo mal y cómo puedo solucionarlo y no volver a repetirlo

Los compiladores pueden detectar determinados problemas sintácticos, semánticos o lógicos e informar de ellos mediante un mensaje de error. Pero este mensaje de error es muchas veces demasiado corto y ambiguo, por lo que puede resultar difícil de interpretar para un desarrollador poco experimentado; otras veces aunque el mensaje del error sea fácil

de descifrar, la causa de este puede ser difícil de encontrar; tanto en un caso como en otro aunque exista un mensaje de error señalando un error el desarrollador puede estar perdido y necesita hacer trazas sobre el programa y cambios que le vayan guiando hasta llegar a la solución del problema, gastando en ello una gran cantidad de tiempo. Para evitar esto, el desarrollador necesita justo en el momento de cometer un error una indicación de lo que está haciendo mal y una indicación de cómo solucionarlo según el contexto en el que se encuentre y debería de ser el propio entorno el que detectase el problema y emitiese las indicaciones.

2.7 Aprendizaje basado en ejemplos y a partir de los errores

Un ejemplo siempre sirve para ilustrar cualquier caso teórico y muchas veces el contexto que proporciona un ejemplo permite comprender mejor el concepto que se pretende transmitir. Sería interesante disponer de ejemplos concretos para los conceptos con los que estamos trabajando, incluso varios ejemplos que correspondan a varios casos; sin embargo, el trabajo con ejemplos es costoso para el que enseña siempre lleva tiempo prepararlos; sería interesante disponer de un sistema que pudiera recoger los ejemplos cuando vayan surgiendo, clasificarlos y relacionarlos con los conceptos.

Si los ejemplos positivos ayudan a realizar un buen aprendizaje, los negativos son aún más adecuados a la hora de proporcionar información de cómo NO se deben hacer las cosas y por tanto, cómo se deberían hacer. Si fuéramos capaces de personalizar estos ejemplos y llevarlos al propio código del desarrollador conseguiríamos explicar al usuario que buenas y malas prácticas está siguiendo a la hora de programar; permitiríamos de esta forma que el usuario estuviese informado sobre sus malos hábitos y mejorase su forma de programar.

2.8 Creación de software de calidad

El trabajo de un ingeniero del software es entregar productos de *calidad* con unos costes y programaciones planificados.

Desde hace mucho tiempo el concepto de calidad de software ha sido algo que muchos autores han intentado definir, por ejemplo Pressman [Pressman, 1997] define *calidad de concordancia* como el grado de cumplimiento de las especificaciones de diseño durante su realización. Lo que parece evidente es que todas las definiciones refieren el concepto de calidad como que el programa implementado *haga lo que tiene que hacer* y además *lo haga bien*. Según Humphrey: *Los productos software deben satisfacer tanto las necesidades funcionales de los usuarios así como realizarlas de una forma segura y consistente* [Humphrey, 1997].

Aunque hay muchos aspectos relacionados con la calidad del software, el primer aspecto de la calidad está relacionado necesariamente con sus defectos. Un defecto, es cualquier cosa que reduce la capacidad de los programas para cumplir completa y efectivamente las necesidades de los usuarios. Los errores triviales de implementación pueden causar serios problemas en el sistema. Según Humphrey, la fuente de muchos defectos software son simples descuidos y errores del programador [Humphrey, 1997].

Para reducir el número de defectos que se introducen en los productos software, el ingeniero debe cambiar la forma de hacer las cosas. Sin embargo, para eliminar los defectos en los productos, sencillamente hay que encontrarlos. La eliminación de defectos es, por lo tanto, un proceso más sencillo que la prevención de defectos.

Los defectos deben de ser encontrados y corregidos para que no lleguen a la fase de producto acabado en explotación. El problema es que lleva mucho tiempo y esfuerzo encontrar y corregir defectos; además, este tiempo es difícil de predecir con lo que a menudo causa problemas de costes y ajuste de tiempos al plan de trabajo. Por tanto, sería muy beneficioso que los ingenieros utilizasen técnicas que evitaran la introducción de defectos.

2.8.1 Comprensión de los defectos

El primer paso para gestionar los defectos es entenderlos. Para ello, hay que reunir datos de los defectos cometidos. A través de estos datos, se podrá comprender como evitarlos y como encontrarlos mejor y corregirlos.

Hay que tener en cuenta los errores que localiza, el propio programador y también los que pueden encontrar otros. Estos últimos errores, tienen gran importancia ya que si el propio programador no los encuentra pueden ser síntomas de debilidades en el proceso de escritura del software por parte del programador.

A la hora de realizar revisiones también es fundamental conocer los tipos de errores cometidos en anteriores programas, teniendo en cuenta que si se desarrolla software de una misma forma los tipos de errores serán muy parecidos a los encontrados anteriormente. Como cada programador tiene una formación y experiencia, diferentes los tipos de defectos también lo podrían ser, por tanto, la estrategia de revisión debería basarse en el perfil de defectos personales de ese programador.

Para conseguir esta calidad en el software es casi imprescindible pasar por un proceso de revisión. Citando a Pressman [Pressman, 1997]: las revisiones del software son un "filtro" para el proceso de ingeniería del software. Se pueden aplicar en varios momentos del desarrollo del software y sirven para detectar defectos que puedan así ser eliminados.

De nuevo, aquí, se pone de manifiesto la importancia de la colaboración en un equipo de desarrollo; una revisión es una forma de aprovechar la diversidad de un grupo de personas para: señalar la necesidad de mejoras en una aplicación, conseguir un trabajo técnico de una calidad más uniforme, o al menos más *predecible*, que la que puede ser conseguida sin revisiones. El mismo Pressman, plantea revisiones en la fase de diseño de la aplicación como un método efectivo para detectar defectos del software en fases tempranas.

A pesar de todas estas recomendaciones, lo cierto es que muchas veces los defectos no se pueden detectar hasta fases donde se disponga de código para realizar pruebas. Es más, las metodologías ágiles de desarrollo de software retrasan, a propósito esta revisión a la fase de codificación. Programación Extrema [Beck, 2000], por ejemplo, propone la *programación en parejas* como medio de asegurar que una persona realiza la revisión del código en el mismo momento en que la otra lo escribe.

Llegados a este punto parece evidente la necesidad de la revisión del código. Este proceso de revisión no siempre cumple las funciones pretendidas por tres razones:

- El programador se conforma básicamente con evitar los errores de compilación, por esto se recomienda que sea otra persona la que revise el código. Como dicen Freedman y Weinberg [Freedman, 1990]: "aunque la gente es buena descubriendo algunos de sus propios errores, algunas clases de errores se le pasan por alto más fácilmente al que los origina que a otras personas".
- El método clásico de revisión de código es el de inspección visual por parte de revisores diferentes al programador. Este método no es seguro ya que el revisar línea a línea se convierte en un proceso tedioso cuando los archivos son grandes,

con lo que el proceso de revisión puede perder su efectividad debido a la perdida de atención que sufre el revisor cuando lleva un tiempo largo realizando la revisión.

- Existen herramientas automáticas de revisión de código, que en general se basan en la utilización de patrones de error, revisión del estilo del código, etc.; pero que sin embargo al final requieren una revisión manual para verificar la existencia de defectos en el código.

Unas buenas habilidades para la depuración incluyen un conocimiento de las causas comunes de errores y como solucionarlos. Los *patrones de error* [Allen, 2002] son relaciones recurrentes entre errores indicados y errores subyacentes en el programa. El conocimiento de esto patrones y sus síntomas ayuda al programador a identificar rápidamente nuevas apariciones de los errores, y además evitarlos sobre la marcha. Los programadores pueden tardar mucho tiempo en reconocer estos patrones si lo dejamos a su experiencia individual. Pero si identificamos estos patrones y los enseñamos explícitamente, podemos ayudar a mejorar la experiencia de los programadores. Además, una vez aprendidos, permiten una identificación y comunicación explícita y los desarrolladores pueden beneficiarse mutuamente de la experiencia de los demás en depuración, y de este modo, adquirir la destreza más rápido que de cualquier otro modo.

2.9 Creación de un modelo para la buena codificación y depuración

Como dice Allen [Allen, 2002] el diagnóstico de errores software tiene mucho en común con la realización de un experimento científico. En ambos casos, debemos seguir los siguientes pasos:

1. Observar el sistema bajo investigación.
2. Formular una hipótesis para explicar las observaciones.
3. Probar esta hipótesis con nuevos experimentos.
4. Repetir hasta que lleguemos a una hipótesis que explique todo el comportamiento observado.

La parte más importante del proceso de depuración y pruebas (y que muchas veces nos saltamos) es el desarrollo de hipótesis de por qué el sistema se comporta como lo hace. Para hacer esto correctamente, es necesario crear un modelo de cómo cada componente del sistema interactúa con otros. Algunos de estos componentes serán modelados sólo a alto nivel. Otros (los que se consideran causa directa del error) serán modelados con bastante precisión. Por ejemplo, al ver los errores de un proyecto con veinte clases planteamos la hipótesis de que clases es la responsable y nos fijamos en detalle sobre esa clase, sin mirar en profundidad el resto. Cuando se realizan más pruebas, es esencial que el programador actualice su modelo teniendo en cuenta los resultados.

La habilidad de los expertos en cualquier campo no es sólo resultado de la inteligencia pura. También es resultado de la experiencia. La experiencia proporciona muchos ejemplos específicos que se pueden generalizar en patrones. Los expertos aplican estos patrones a nuevas situaciones y así obtienen razonamientos más efectivos. En particular, ignoran detalles irrelevantes más rápidamente y se centran en lo que es importante. Muchas veces es posible identificar *patrones de error* con sus síntomas, causas y soluciones. Si se es consciente de la existencia de un patrón de error, se pueden identificar las ocurrencias de ese patrón más rápidamente y solucionarlo. Examinando estos patrones de error podemos hacer más

que identificarlos, podemos también que prácticas de codificación pueden ayudar a prevenir la aparición de este error en el futuro.

El problema reside en que la creación y el aprendizaje de los patrones de error son complicadas y requieren de un tiempo que muchas veces no se dispone. Ante esto surgen ideas de intercambio de información sobre errores en entornos colaborativos, de tal manera que cualquier programador pueda consultar información un error y si no existe introducir la información que tenga en ese momento.

2.10 Adquisición de competencias para eliminar / evitar defectos en el software

Para conseguir lo planteado en la sección anterior es imprescindible mejorar el proceso de adquisición de habilidades por parte de los futuros ingenieros de software. Sabemos que es muy importante tener programadores con experiencia ya que *la calidad y seguridad del software producido depende inevitablemente de la destreza y experiencia de los programadores involucrados*. Además, *Para formar a más programadores es insuficiente la enseñanza tradicional de conocimiento teórico. Necesitamos transmitir habilidades prácticas en el desarrollo de sistemas robustos, esto lleva años de experiencia* [Allen, 2002]. Esto enlaza directamente con el aprendizaje basado en competencias que se plantea con el Espacio Europeo de Educación Superior y que trata en profundidad el "Libro blanco del Título de grado en Ingeniería Informática" elaborado por un grupo de trabajo de la Conferencia de Directores de Ingeniería Informática y editado por la ANECA [ANECA 2005].

Dentro de la competencia específica definida como "Programación" consideramos fundamental el conocimiento práctico que consiste en la habilidad de diagnosticar eficientemente y subsanar los errores del sistema software. La depuración efectiva está lejos de ser una habilidad trivial. Buscar y eliminar errores ocupa una porción significativa del tiempo de desarrollo en un proyecto software. Si esta tarea se pudiera realizar de forma más eficiente, el software resultante podría ser más fiable y el proceso de desarrollo sería más rápido. Por tanto, es fundamental el aprendizaje no sólo de técnicas de codificación sino también de detección y corrección de errores. El aprendizaje de estas técnicas no es sencillo y para alcanzar un dominio adecuado es imprescindible complementar los conocimientos teóricos con prácticas que ayuden a la buena asimilación de estos conceptos.

2.11 Conclusión

Es necesario un entorno "plug-and-play" que pueda utilizarse desde cualquier ordenador sin necesidad de instalación de software; simplemente conectarse a Internet. Un entorno Web también facilitaría la integración dentro otros sistemas Web pensados para el aprendizaje de la programación y sería un paso decisivo que posibilitaría una mejor adquisición de buenas prácticas de programación.

Un modelo que ha tenido éxito en otros campos es el **aprendizaje basado en errores**. En el campo de la programación se ha explotado poco este enfoque; pese a que los compiladores generan determinados errores a la hora de compilar. Estos **mensajes de error** son utilizados por los programadores para corregir el error concreto de ese programa; pero **no se aprovechan como una oportunidad para aprender** y evitar que este error vuelva a ocurrir.

Por otra parte, los **mensajes de error de los compiladores son excesivamente escuetos y en ocasiones crípticos** por lo que muchas veces es necesario tener una experiencia para

poder utilizar correctamente la información proporcionada por el error y corregirlo de una forma efectiva. Sería muy útil acceder a una base de conocimientos que proporcionase información sobre el mensaje de error: una explicación más amplia, en qué contextos se puede dar el error, cuál es la causa y cómo se puede solucionar. Sería como si un programador novato tuviera un grado suplementario de experiencia.

Si el sistema puede realizar una detección automática de problemas y puede dirigir al desarrollador hacia la información adecuada para saber cual es el problema y poder solucionarlo estaremos reduciendo la necesidad de una comunicación asíncrona entre desarrolladores o entre profesor – estudiante en un entorno de aprendizaje con lo cual los usuarios podrán realizar auténtico trabajo autónomo y con el ritmo adecuado; pero a la vez respaldado por el apoyo de un tutor automático.