B Programación ágil

A la hora de realizar un proyecto software de dimensiones considerables, se hace necesario la utilización de técnicas y metodologías que faciliten el desarrollo del mismo así como asegurar un código eficiente, libre de errores y bien documentado. Con esto además, se persigue reducir el tiempo invertido en obtener el producto final.

Los apartados siguientes hacen mención a las metodologías utilizadas en el PFC, y los beneficios que brindan al proyecto.

B.0.4 Diseño de software por módulos

La programación modular es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable. Al aplicar la programación modular, un problema complejo debe ser dividido en varios subproblemas más simples, y estos a su vez en otros más simples. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación. Ésta técnica se llama refinamiento sucesivo, divide y vencerás ó análisis descendente (*Top-Down*).

Cuando se crea una aplicación modular en vez de una monolítica, se construyen numerosos módulos independientes (a veces incluso compilados por separado) con el fin de que sean fáciles de implementar. Una vez creados, se les hace trabajar juntos con lo que se consigue realizar el ejecutable deseado.

Esto hace a los sistemas modulares mucho más reutilizables, puesto que muchos de dichos módulos pueden utilizarse directamente en otros proyectos. Estos sistemas tienen también la ventaja de que son más fáciles de construir para un equipo de programadores ya que cada miembro se puede concentrar en el desarrollo de un módulo independiente y no recae en nadie la tarea de crear un sistema complejo entero, lo que mejora la productividad.

Para que cada módulo sea independiente del resto se utilizan distintos métodos que se comentarán en los siguientes apartados, y para poder comunicarse unos con otros se deben definir unas interfaces con las que manejarlos.

Ahora viene el problema del refinamiento sucesivo: ¿Cuándo parar la descomposición en problemas más simples?. Si el refinamiento es excesivo podría dar lugar a un número tan grande de módulos que haría poco práctica la descomposición. Hay que llegar entonces a un compromiso entre número de módulos resultantes, y tareas claramente independientes separadas en módulos.

B.0.5 Principios S.O.L.I.D.

Cada sigla representa un principio, y hablan del diseño orientado a objetos en términos de la gestión de dependencias. Las dependencias entre unas clases y otras son las que hacen al código más frágil o más robusto y reutilizable. Dichos principios son:

- Single Responsibility Principle (SRP): Viene a decir que cada clase debe de tener una única responsabilidad, es decir, debe de ocuparse de una sola tarea. Visto de otro modo, cada clase debe de tener sólo un motivo por el cual tenga que ser modificada. Como contraejemplo, se propone la típica clase *saco* en dónde se van implementando todos los métodos que son útiles (por ejemplo funciones de cálculo de áreas, convertir unidades, etc.) pero el programador no sabe dónde ponerlas; la clase *saco* por tanto se encargara de más de una tarea (tiene más de una responsabilidad), por lo que deriva en una clase muy extensa, con métodos diversos y poco organizados.

  El resultado de seguir SRP es la creación de clases con nombres muy descriptivos, con muy pocos métodos y cada método con menos de 15 líneas de codigo (idealmente). Por encima de este límite, se entiende que posiblemente la función se ocupe de más tareas de las necesarias, pudiendo refactorizar así el código.

- Open-Closed Principle (OCP): En este caso se insta a que una entidad software (clases, modulos o funciones) se encuentren abiertas a extensiones pero cerrada a modificaciones. Esto quiere decir que se debe poder alterar el funcionamiento sin tener que modificar su propio código fuente ya que, de lo contrario, modificaciones en el código de alguna de las entidades pueden generar efectos indeseables en cascada.

  Hay varias técnicas para conseguir este objetivo, si bien las más utilizadas serian la herencia y redefinición de los métodos de la clase padre, o inyección de dependencias que cumplen el mismo contrato (que tienen la misma interfaz pero implementan distinto funcionamiento).

- Liskov Substitution Principle (LSP): Está estrechamente relacionado con el anterior en cuanto a la extensibilidad de las clases cuando ésta se realiza mediante herencia o subtipos. Lo que viene diciendo es que si una función recibe un objeto como parámetro, de tipo X y en su lugar se le pasa otro de tipo Y, que hereda de X, dicha función debe proceder correctamente. La cuestión por tanto es si la función de verdad está diseñada para hacer lo que debe, aunque quien recibe como parámetro no sea exactamente X, sino Y.

- Interface Segregation Principle (ISP): Cuando se emplea el SRP también se implementa el ISP como efecto colateral. El ISP defiende que no hay que obligar a los clientes (otras clases o entidades) a depender de clases o interfaces que no necesitan utilizar. Tal imposición se da cuando una clase o interfaz dispone de más métodos de los que una entindad cliente necesita para sí misma, convirtiéndose en dependiente y multiplicando la posibilidad de catástrofe frente a cambios de la interfaz o clase base.

- Dependency Inversion Principle (DIP): La inversión de dependencias da origen a la conocida inyección de dependencias, una de las mejores técnicas para lidiar con las colaboraciones entre clases, produciendo un codigo reutilizable, sobrio y preparado para cambiar sin producir efectos *bola de nieve*. DIP explica que un módulo concreto A, no debe depender directamente de otro módulo concreto B, sino de una abstracción de B. Tal abstracción es una interfaz o una clase (que podría ser abstracta) que sirve de base para un conjunto de clases hijas.

B.0.6 TDD (Test Driven Development)

Desarrollo guiado por pruebas, o Test Driven Development, es una práctica de programación ágil que involucra otras dos prácticas: Escribir las pruebas primero (*Test First Development*) y Refactorización (*Refactoring*). TDD facilita un diseño mantenible a través de la noción de pruebas. Estas pruebas obligan a reflexionar sobre el comportamiento del código y la forma de garantizar que funciona según lo previsto. La mayoría de las veces, el código influenciado por TDD es relativamente seguro, y sin duda, es bastante simple. Un código seguro y simple es más fácil de cambiar que uno complejo y frágil. Dado que en TDD el código se respalda con pruebas, cualquier cambio que rompe el código se descubre fácilmente por lo que se ahorra tiempo tanto en depuración como en desarrollo.

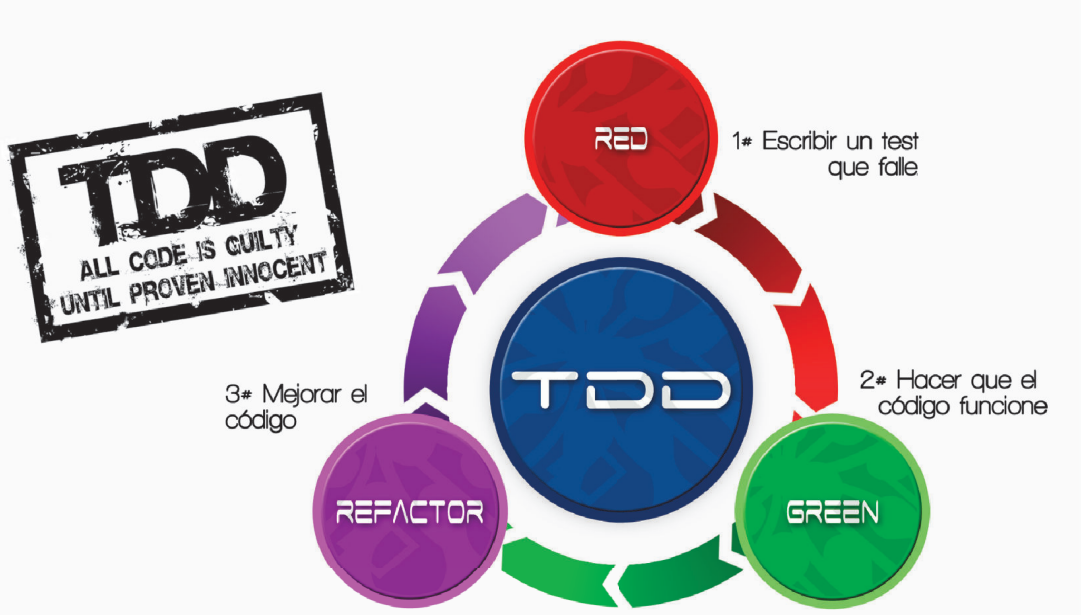


Figura 70: Funcionamiento de TDD

El método a seguir para utilizar esta práctica es, en primer lugar, realizar un test (sin importarnos el que todavía no esté implementado el código que va a testar). En segundo lugar, se implementará el código justo y necesario (no más) para hacer que pase el test. Con esto se consigue un código limpio, sin relleno, evitando así entre otras cosas funcionalidades extra que no son necesarias en el proyecto.

Acostumbrarse a TDD no es fácil, pero consigue reducir el tiempo invertido en terminar el proyecto. Así mismo, reduce notablemente el tiempo de corrección de errores, ya que las pruebas se encargan de delatar las entidades que está fallando.

Implementar correctamente TDD implica intentar que el 100% del código disponga de un test que verifique su correcto funcionamiento. En la mayoría de los casos existen situaciones que son imposibles de testar o mejor dicho obligan a invertir demasiado código y tiempo sólo para el test, por lo que esos casos especiales se implementarán o no según su relevancia y criticidad. Cabe comentar que el código de terceros (librerías ya hechas, etc.) no hay que testarlas puesto que se entienden como cajas negras que funcionan correctamente.

B.0.7 BDD (*Behaviour Driven Development*)

Es una técnica de programación que cuestiona el comportamiento de una aplicación antes y durante el proceso de desarrollo. Mediante el enfoque en el comportamiento de las aplicaciones, los desarrolladores intentan crear un lenguaje común entre todos: gestión, usuarios, desarrolladores, jefe de proyecto y expertos de negocio.

La idea es que las palabras usadas en el código, tienen un poder enorme de confundir o de facilitar el trabajo al desarrollador. Por lo tanto, se tiene que hacer un uso eficiente de la lengua: Los test deben tener nombres que identifiquen lo que están probando en realidad, es decir, el nombre de los test deben ser frases que se entiendan y que digan lo que hacen, en detalle.

B.0.8 Patrones de diseño

Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software. En otras palabras, los patrones de diseño brindan una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos similares. Utilizando patrones de diseño se mejoran características de los diseños, como la cohesión y el acoplamiento.

Los patrones no ofrecen implementaciones concretas para solucionar un problema. Más bien nos ofrecen una estructura para relacionar clases y objetos. Dado que existen similitudes entre el diseño guiado por modelos y el diseño de software, muchos de estos patrones son igualmente válidos y aplicables.

A continuación se resumen los patrones de diseño más usuales y que son aplicables al diseño guiado por modelos (En varios modelos y diagramas serán aplicables varios patrones de diseño. A pesar de que en el mundo real se deban aplicar siempre que sean posible, en estos no se hará por simplificación de los diagramas/modelos).

Los patrones de diseño pretenden:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores. Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Asimismo, no pretenden:

- Imponer ciertas alternativas de diseño frente a otras.
- Eliminar la creatividad inherente al proceso de diseño.
- No es obligatorio utilizar los patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. ”Abusar o forzar el uso de los patrones puede ser un error”.

Los patrones de diseño se pueden clasificar en tres grupos fundamentales:

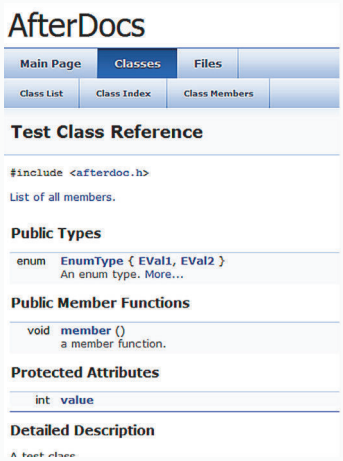
- Patrones estructurales: Los patrones estructurales describen cómo utilizar estructuras de datos complejas a partir de elementos más simples. Sirven para crear las interconexiones entre los distintos objetos/métodos y que estas relaciones no se vean afectadas por cambios en los requisitos del sistema.
  - Fachada (*Facade*).
  - Adaptador (*Adapter*).

- Patrones creacionales: Los patrones creacionales nos ayudan a gestionar la creación de objetos.
  - Instancia única (*Singleton*). Muy importante para este proyecto, ya que ofrece la posibilidad de que entre todas las aplicaciones pueda existir únicamente un objeto de cierta clase.
- Patrones de comportamiento: Fundamentalmente explican el comportamiento entre objetos/subsistemas de nuestro programa/modelo.
  - Estrategia (*Strategy*).
  - Plantilla (*Template*).
  - Estado (*State*).

B.0.9 Documentación

Es una de las partes más importantes a tener en cuenta en el desarrollo de software. La principal causa de un llevar a cabo un proyecto a buen ritmo es la reutilización de código, ya que si la funcionalidad buscada existe y está probada, ahorra tiempo y trabajo. Es por ello que un requisito vital para llevar a cabo ésta tarea es entender el código que se va a reutilizar. Puesto que uno no sabe a priori si va a necesitar el código que está desarrollando, o si otra persona ajena al proyecto en ese momento se incorpora o necesita utilizar nuestro código, es necesario documentar correctamente todo el proyecto.

Existen herramientas que ayudan a crear una buena documentación, como por ejemplo Doxygen. Es una herramienta muy potente que convierte los comentarios (bien formateados según unas reglas para poder captarlos), en un manual web, bien estructurado y sencillo de entender.




```
/*! A test class */

class Test
{
  public:
    /** An enum type.
     *  The documentation block cannot be put after the enum!
     */
    enum EnumType
    {
      int EVal1,     /**< enum value 1 */
      int EVal2      /**< enum value 2 */
    };
    void member();   //!< a member function.

  protected:
    int value;       /*!< an integer value */
};
```

Figura 71: Izq: Ejemplo de manual generado por Doxygen. Dcha: Código comentado adecuadamente.

B.0.10 Compilación

Aunque los entornos de desarrollo comunes como Eclipse, Visual Studio, etc. disponen de su propia funcionalidad para llamar al compilador, solamente hacen eso: compilar las cabeceras y código fuente de la estructura del proyecto.

Ciertamente, para proyectos pequeños suele ser suficiente, pero existen otras tareas a tener en cuenta a la hora de compilar que se pueden automatizar, lo que no se contempla en los IDE normalmente. Se habla por ejemplo de la capacidad de programar tareas antes y después de la compilación, como copiar las librerías de las que se dependa directamente de los directorios dónde los demás proyectos las han compilado, y así asegurarse que siempre se copia la última versión actualizada de la librería. Otras tareas importantes como mantener una estructura de directorios personalizada no puede llevarse a cabo en los entornos habituales, o ejecutar scripts y/o aplicaciones adicionales.

Todas las tareas descritas y más, se pueden realizar gracias a herramientas de compilación externas, que permiten al programador automatizar procesos gracias a la utilización de scripting (bien con lenguajes conocidos o uno específico).

De entre las herramientas propuestas en un principio a utilizar en el proyecto (a saber: MAVEN, ANT, CMAKE y SCONS), se escogió ANT después de ver las siguientes comparativas. MAVEN está pensado únicamente para proyectos JAVA por lo que se descarta directamente.

	SCONS	CMAKE
Dependencias	Scons y Python dependen entre si y sus actualizaciones pueden hacer inservibles los scripts creados anteriormente	Cmake sólo depende de C++
Extensibilidad	Con Python se puede extender la funcionalidad todo lo que se desee	Permite introducir nuevos comandos a través de una instrucción de su propio lenguaje
Velocidad	Más lento en grandes proyectos	Mayor rapidez de ejecución
Tareas comunes	Más cantidad de código	Implementación sencilla
Estabilidad	-	Más estable y robusto
Aprendizaje	Hay que aprender Python	Sintaxis sencilla
Plataforma	Funciona bien en Linux	Funciona bien en Linux y Windows
Documentación	Muy buena, distintos formatos, bien estructurada y gran comunidad de desarrolladores	Buena, bien estructurada y gran comunidad de desarrolladores

Debido a la gran dependencia de SCONS con las versiones de python, y teniendo en cuenta que es más lento que CMAKE, se descarta a favor de éste último.

	ANT	CMAKE
Dependencias	Es necesario instalar la JVM	Cmake sólo depende de C++
Scripting	XML (Aporta mayor legibilidad)	Lenguaje propio (Heredado de versiones anteriores de make y autoconf)
Plataforma	Cross-platform (JVM)	Linux y Windows

Extensibilidad	Permite definir etiquetas nuevas para aumentar su funcionalidad	Permite introducir nuevos comandos a través de una instrucción de su propio lenguaje
Documentación	Poco documentado para C++	Buena, bien estructurada y gran comunidad de desarrolladores
Otros	-	Recomendado para GoogleTest y otras tecnologías

Como solución final se implementó ANT frente a CMAKE dada la facilidad de integración con Eclipse, mayor facilidad de uso y aprendizaje, y la posibilidad de poder invocar a CMAKE u otra herramienta en caso de necesitar alguna funcionalidad específica no contemplada en ANT de forma sencilla.

B.0.11 Entorno de desarrollo

La arquitectura seleccionada para llevar a cabo el proyecto se compone de una workstation con Ubuntu 10.4 LTS, para la edición de código en C++; un repositorio en red, para el control de versiones del código; una máquina de compilación, encargada de compilar periódicamente la última versión del proyecto disponible en el repositorio y comprobar su correcta compilación; una máquina de referencia, que en un proyecto comercial hace las veces del equipo destinado a utilizar la aplicación y es dónde la máquina de compilación ejecuta los test; y un equipo Hudson para controlar las dos máquina anteriores.

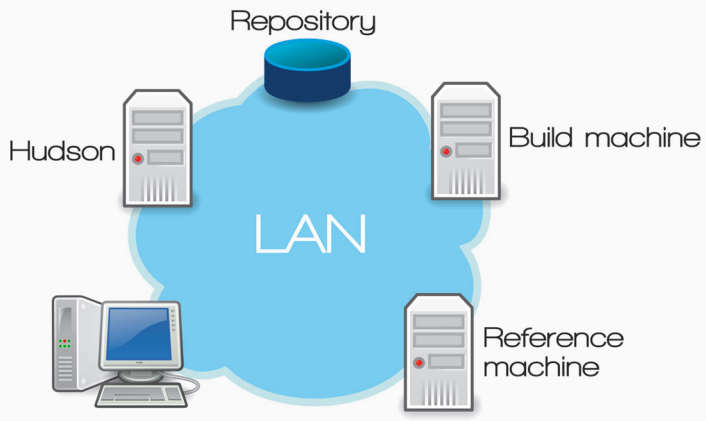


Figura 72: Arquitectura de red para el entorno de desarrollo.

El despliegue de hardware utilizado conlleva un rendimiento mayor a la hora de desarrollar software, ya que automatizando ciertas tareas, se detectan y corrigen los errores con mayor facilidad, pudiendo avanzar más rápidamente.

Los equipos y el software necesario para la realización del proyecto se detalla como sigue:

Máquina de desarrollo.

Es una workstation dónde se ha instalado el IDE de software libre Eclipse Helios, con el plugin CDT para la composición de código en C++ (ya que originalmente Eclipse se utiliza para entornos Java). Para tener un mayor control a la hora de compilar, tal y como se ha explicado, se ha optado por utilizar la herramienta de compilación ANT en vez del propio Eclipse, ya que otorga mediante scripting mayores posibilidades que simplemente compilar código.

Además, para la implementación de TDD se han instalado las librerías de Google Test y Google Mock, ambas de libre distribución y código abierto. La primera permite la creación sencilla de tests, poniendo al servicio del programador multitud de macros y funciones para monitorizar los resultados de la ejecución del código testado.

La segunda librería, permite la creación de lo que comúnmente se denominan “mocks”, que vienen a ser objetos de las clases del código fuente que falsifican su funcionamiento pero presentan al resto del test la misma interfaz y comportamiento. En otras palabras,

se tiene un objeto "comodín" con la apariencia del objeto real, pero vacío en código, al que se le especifican los resultados que tiene que devolver al llamar a sus funciones falsificadas. Esto permite probar clases que de otra forma no permiten ser testadas, como por ejemplo clases que utilicen en sus métodos una llamada a una base de datos que no esté disponible o que en el momento de crear la clase todavía no exista. Mediante la creación de un mock, se puede falsificar el funcionamiento de ése método que accede a la base de datos, y hacer que devuelva siempre el valor deseado para poder seguir testando el resto del código.

Para cualquier aplicación que se precie, debe existir la capacidad de llevar un historial de ejecución, o lo que es lo mismo, un log. Mediante la creación de logs, se puede seguir la ejecución del programa viendo en qué funciones entra, cuando sale, errores del aplicativo, avisos para el usuario, o mensajes para el desarrollador. Para el HMI se buscó directamente librerías de log de software libre que fuesen populares, y resultó ser Log4cplus, la cual se encuentra bien documentada (de hecho mediante Doxygen), y al ser de libre distribución dispone de gran aceptación por parte de los usuarios por lo que es relativamente fácil encontrar ejemplos, soluciones a posibles problemas, etc.

Por último, la aplicación o librería creada debe tener capacidad de configuración por parte del usuario, lo que se consigue mediante ficheros que contienen información relevante al funcionamiento y que se pueden cambiar según las necesidades. En el caso de éste proyecto, se configura la aplicación al inicio, teniendo efecto nulo en la ejecución actual el cambio de algún parámetro del fichero de configuración, para lo que habrá que reiniciar el aplicativo.

Según el tipo de aplicación, se ha utilizado Rudeconfig para la lectura de ficheros de texto plano que contienen los parámetros de configuración, o mediante las librerías Qt, que ofrece el objeto *QSettings* el cual dispone de métodos ya implementados para acceder y almacenar fácilmente la configuración albergada en un fichero.

Repositorio en red

Un repositorio no es más que una zona de almacenamiento donde guardar el código fuente del proyecto. La funcionalidad que lo hace necesario es la capacidad de almacenar las distintas versiones del código a medida que se va avanzando en el desarrollo. En principio no existe ninguna necesidad de tener que implementar el repositorio en red, dado que se puede ubicar en cualquier carpeta local a elección del programador. La

necesidad de estar en red, viene entonces por el uso de metodologías ágiles en grupo, ya que se busca que varias personas puedan acceder al código bien para reutilizarlo en sus proyectos o bien para colaborar en su desarrollo, pero garantizando una correcta gestión de los datos para no perderlos ni reemplazarlos por otros que sean erróneos.

Cada equipo que vaya a colaborar en el desarrollo de un proyecto ubicado en el repositorio de red, tiene que instalarse un software cliente para acceder al repositorio. Este tipo de software, permite clonar los datos remotos a una carpeta en el disco duro local, además de llevar un control de todos los cambios que afecten a los ficheros y estructura de directorios dentro de dicha carpeta para poder actualizar la versión *online* y que el resto de participantes puedan beneficiarse de los cambios. Para ello, después de modificar o añadir archivos, se requiere hacer lo que se denomina un *commit*, que es escribir una descripción de los cambios realizados. El software de control de versiones relaciona dicha descripción con los cambios realizados y así poder separar cada versión subida por los usuarios.

Es deber del usuario mirar si ha habido cambios en el repositorio antes de subir su versión del proyecto. En caso de haber cambios, significa que otro programador ha subido modificaciones, por lo que se deberán bajar las nuevas, volver a compilar el código y hacer pasar los tests para comprobar que todo sigue funcionando correctamente al haber añadido nuestros cambios. Una vez realizadas las comprobaciones y/o modificaciones oportunas, se sube al repositorio remoto con lo que los demás participantes podrán ver una nueva versión, y así continuamente.

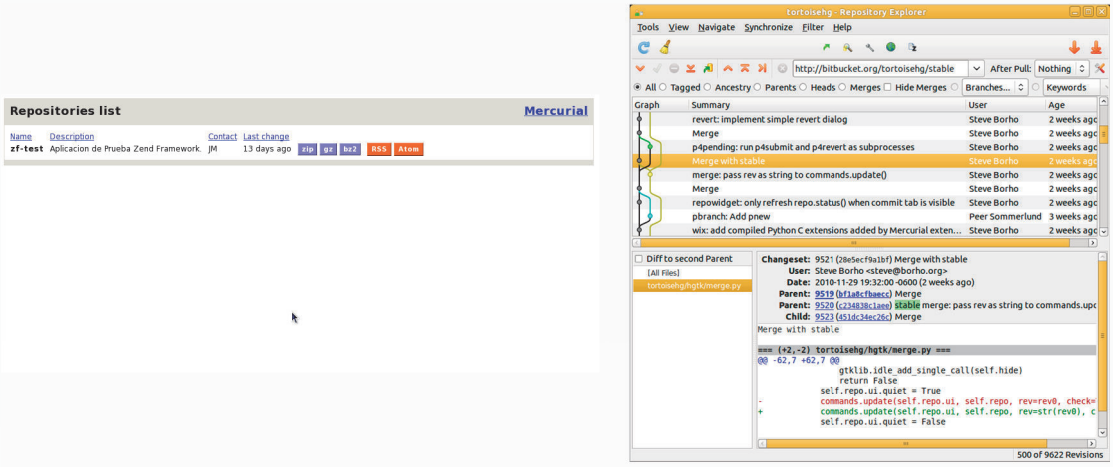


Figura 73: Izq: Interfaz de Mercurial. Dcha: Interfaz de TortoiseHg.

Las ventajas de utilizar un repositorio, además de las ya comentadas, es la facilidad de volver a versiones anteriores en caso de haber errores. Para ello es recomendable realizar *commits* muy a menudo, a cada poco cambio que se haga, para que sea más fácil ver en que momento se ha roto el código.

El repositorio elegido para albergar los proyectos, ha sido Mercurial, un sistema de control de versiones distribuido. Otras alternativas igualmente populares son Git y Bazaar, pero descartadas ya que Mercurial se entiende como un motor más sencillo y amigable de configurar y manejar, a lo que Git responde con mayor dificultad ya que permite mayores comandos a bajo nivel que no son necesarios en este caso. Bazaar sin embargo, también se corresponde con un sistema sencillo, pero no concibe la integración de IDE multiplataforma, lo que supone una desventaja ya que se decidió mantener abierta la posibilidad de desarrollar tanto en plataforma Windows como en Linux.

Como cliente Mercurial, se instaló TortoiseHg, ya que es dedicado a dicho sistema de control de versiones y puede correr perfectamente tanto en Windows como en sistemas Linux. La comunicación entre los equipos de desarrollo y el repositorio pueden ser tanto HTTP como SSH, según se requiera más o menos protección en las transacciones (por ejemplo para repositorios en los que haya que salir de la red local).

Máquina de compilación.

Como bien se puede adivinar, es un equipo dedicado exclusivamente a la compilación de proyectos. Es por ello que incluso no sería necesaria la instalación de un sistema operativo con interfaz gráfica, pudiendo administrarse únicamente desde consola.

La máquina únicamente deberá tener, además de lo necesario para poder administrarla, el software y librerías necesarias para llevar a cabo la compilación. Esto entiende una configuración igual que el equipo de desarrollo, teniendo las mismas versiones de librerías, por lo que cualquier actualización en el equipo de desarrollo deberá realizarse en la máquina de compilación.

Una pregunta normal seria: ¿y para qué utilizar una máquina para compilar si en el equipo de desarrollo ya hay que compilar el proyecto?. La respuesta se encuentra en la necesidad de automatizar el proceso de compilación y testado de código para la detección precoz de errores en una máquina que tiene lo justo para hacer funcionar el aplicativo, lo que no es así en el equipo de desarrollo ya que suele estar mucho más equipado, lo que puede llegar a encubrir errores en el código por dependencias no con-

templadas.

Máquina de referencia.

Es una copia del tipo de máquina que finalmente será la que ejecute la aplicación, por lo que es un sistema limpio. Esto significa que dispone de todavía menos software instalado que la máquina de compilación, teniendo únicamente las librerías necesarias para poder ejecutar el archivo binario del aplicativo sin problemas, además de los test que comprobarán no solamente que la aplicación está correctamente construida, sino además testarán su integración con el entorno en el que se debe ejecutar.

Máquina Hudson.

En las técnicas programación ágil, se hace necesaria una herramienta de integración continua. Esto no es más que lo ya explicado: realizar "integraciones automáticas" de un proyecto lo más a menudo posible para detectar los fallos cuanto antes. Este proceso comprende la compilación y la ejecución de los test de un proyecto.

Es aquí dónde entra en juego una nueva herramienta: Hudson. Es una aplicación que se encarga de realizar automática y periódicamente las tareas de compilación y ejecución de test. Se configura para que se conecte a la máquina de compilación y referencia, se descargue en la máquina de compilación la última versión de los proyectos y los compile, para posteriormente copiar los archivos binarios (aplicación y tests) en la máquina de referencia, ejecutarlos, y comprobar que funcionaron correctamente.

Figura 74: A riesgo de parecer inapropiado, denotar que en el mundo de la programación es muy común encontrar chistes para promocionar las buenas prácticas.

Como resultado, Hudson ofrece información mediante una interfaz web en la que se puede observar todas las iteraciones realizadas hasta el momento en cada uno de los proyectos. En cuanto un proceso de compilación o ejecución de test falla, remarca el estado del proyecto como fallido, en cuyo caso se debe dejar lo que se está haciendo y subsanar el error. Esto es, darle máxima prioridad a Hudson, ya que el resultado correcto de sus iteraciones es garantía de funcionamiento del proyecto, y no tomar en consideración sus resultados y seguir avanzando en el código seguramente ocasiona errores graves y difíciles de reparar rápidamente.

Es por ello que normalmente existen numerosas bromas que arremeten contra la incorrecta compilación de código, buscando por medio de hacer gracia que el programador se acuerde de estar pendiente de los resultados y así además promocionar las técnicas de desarrollo ágil para un software de calidad.