

Ingeniería del Software Orientada a Objetos

La Ingeniería del Software y la Orientación a Objetos son dos áreas cuya intersección produce un amplio abanico de técnicas y metodologías que pretenden facilitar la construcción de software. Este artículo revisa algunas de estas técnicas, que pueden ser de gran utilidad para el desarrollo de proyectos complejos con éxito.

Introducción

No muchos años atrás, cuando el que escribe aún estudiaba en la Facultad de Informática, la orientación a objetos (OO) se presentaba como una técnica novedosa y revolucionaria. En estos tiempos que corren de sistemas de ventanas, multimedia, sistemas multicapa en red y lenguajes de alto nivel, nadie se cuestiona el hecho que aquella afirmación no cayó en el olvido como ya ocurrió con tantas otras.

La OO se basa en tres principios básicos: todo son objetos, encapsulamiento / ocultación y herencia / polimorfismo. El primer principio indica la unidad básica de trabajo. El segundo permite englobar en un mismo concepto a los datos y a las operaciones. El tercero permite agrupar y tratar de igual forma a objetos similares. Tras esta simplicidad, la metodología de desarrollo OO incluía entre sus bondades una muy prometedora: permite abarcar las siguientes fases de un proyecto software: Por un lado Análisis y Diseño (OOAD) y por otro Programación (POO). Ya por aquel entonces, se recalca la necesidad de usar lenguajes de modelado para desarrollar proyectos: OOSE, OMT-2, Booch'93 o UML. Este último, fruto de la fusión y de mejoras de los anteriores, aún estaba en proceso de desarrollo en *Rational* (compañía integrada en IBM hoy en día).

Aun con la cotidianeidad del paradigma orientado a objetos, muchos informáticos sumergidos en un mar de siglas se realizan la siguiente pregunta:

“Pues muy bien, ya sé UML... ¿y ahora qué?”

UML es un lenguaje de especificación, visualización, construcción y documentación de propósito general, aunque especializado en sistemas software. Por su propia definición, la mayoría de autores no se pronuncian tajantemente acerca de cómo debe usarse UML; por lo mismo que nadie se atrevería a limitar de forma general el uso de un bucle *for*. Sirvan como contraejemplo las sugerencias de uso de Pierre Alain Muller (1997):

Actividad	Diagrama UML
Análisis de Requisitos	Casos de Uso
Para cada <i>escenario</i> de caso de uso	Secuencia
Refinamiento	Colaboración
Diseño y arquitectura	Clases
Refinamiento de operaciones complejas	Actividad Estados
Implantación e instalación	Organización

Debido a la magnitud de los sistemas actuales y al tamaño de los equipos de desarrollo, la construcción de todo producto software (proyecto) pasa por una serie de fases. Estas son habitualmente: análisis, diseño,

Las
herramientas de
IS (CASE) deben
adaptarse
a nuestros
procesos.

implementación, testing, instalación y configuración.

Para abordar proyectos de semejante envergadura, el trabajo se divide en tareas más pequeñas: el sistema se divide en subsistemas. El desarrollo de cada fase se puede dividir en procesos, entendiendo como tales a secuencias de actividades que hay que completar para alcanzar un objetivo. En la Ingeniería del Software Orientada a Objetos, la OO se puede utilizar durante todo el proceso en global.

Pero no basta con una metodología de desarrollo y un lenguaje de modelado e implementación. Para aplicar la OO y UML de forma efectiva es conveniente valorar la utilidad de algunas técnicas y ciclos de vida del software. A continuación se presentan algunas claves para controlar el orden interno en que se completan los proyectos y evitar la pérdida de información durante la transición entre fases.

Los Ciclos de Vida

El desarrollo de software no es fácil: las técnicas son relativamente nuevas, los sistemas actuales son complejos y los requisitos de los clientes cambian con bastante más frecuencia de lo que muchos desearíamos. Cualquier organización que desarrolle software debería plantearse qué proceso le interesa seguir para la realización de sus productos, ya sea para todos o para cada uno en particular.

En los últimos años, una tecnología orientada a resolver este problema que ha sonado con mucha fuerza es la gestión de *workflows*. De forma más concreta, el Proceso Unificado de Rational (RUP) y la Programación eXtrema (XP) son dos de las aproximaciones al proceso de desarrollo más populares. Démosle un rápido vistazo a los ciclos de vida tradicionales antes de repasar estas técnicas.

1. Ciclos Tradicionales

Con el primer ciclo tradicional con el que nos encontramos es con el ciclo de vida Clásico o en Cascada:

Análisis→Diseño→Implementación→Testing

Ingeniería del Software Orientada a Objetos

En este ciclo siempre se volvía a la fase anterior para resolver cualquier error. Se le achacó que los proyectos casi nunca siguen un flujo secuencial. El ciclo se modificó para plasmar características más realistas, apareciendo el Modelo en V y el Modelo Real (una versión mejorada de este último, el Modelo W, ya apareció en el artículo de Vos, Nácher y Palacios en el anterior número de actualidadTIC).

Para que los clientes pudiesen ver el producto antes de que estuviese terminado, se introdujeron los ciclos de vida Orientados al Prototipado (un prototipo es una versión incompleta de la aplicación). Existen dos aproximaciones de prototipo posibles: que sea desechable o que se mejore de forma incremental hasta conseguir la versión final del sistema.

Otro de los ciclos, el ciclo de vida *En Espiral*, afronta el proceso de forma iterativa e incremental, añadiendo funcionalidades al sistema progresivamente:

Análisis→Diseño→Implementación→Testing→
Análisis→Diseño...

Tras validar las sucesivas iteraciones con el cliente, el proceso va convergiendo hacia el producto esperado por ambas partes.

2. Los Workflows: ¿Qué iGRFJTX#&! son?

Un *workflow* (flujo de trabajo) es la automatización, completa o parcial, de un proceso de negocio. Durante este proceso, las tareas, documentos o información en general pasa de un participante a otro siguiendo una serie de reglas. La misión principal de los workflows es controlar los procesos que se inician en una compañía para atender a una demanda externa. Se utilizan para conseguir que tanto los requisitos y sus actualizaciones como los recursos sean trazables: que su origen y su evolución puedan identificarse.

Normalmente se hace referencia a los workflows por los Sistemas de Gestión de Workflows. Estos son herramientas GroupWare (desarrollo concurrente y remoto, vía web generalmente) que permiten hacer un seguimiento de la evolución del proyecto. No solo definen el proceso sino que aportan un motor que se encarga de realizar las transiciones entre los pasos que han de seguirse para cumplir cualquier tarea (según indique el propio proceso).

Estos sistemas permiten controlar en cada proyecto por qué fases pasan los documentos, asociar comentarios a la información, asignar recursos, representar roles de los participantes, personalizar la información, avisar (mediante mensajes) a los responsables de fechas de entregas de documentación, controlar los privilegios de acceso, etc. MQ Series Workflow de IBM, Changengine de HP, Staffware o Teamware's i-Flow son ejemplos de estos sistemas.

3. Rational: RUP

El Proceso Unificado de Rational (RUP) es iterativo e

incremental, igual que el modelo en espiral. RUP se define mediante la combinación de flujos de trabajo fundamentales (*workflows*) y fases (ver figura). Está compuesto por una serie de filosofías y prácticas, un modelo de procesos, una librería de contenidos y un lenguaje de definición de procesos extensión de UML: SPEM.

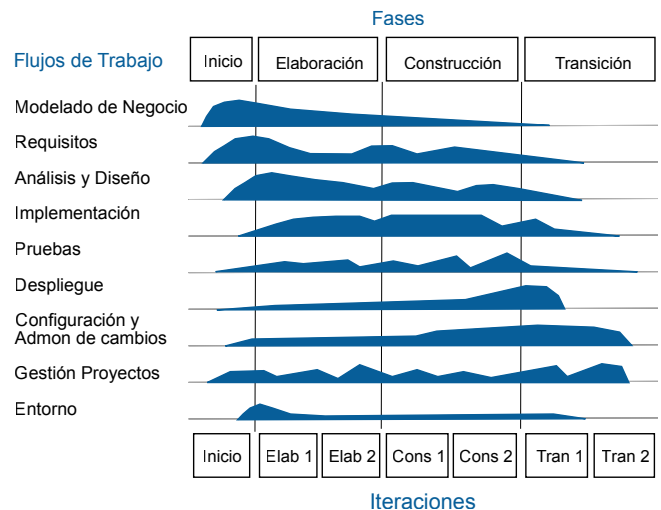


Figura 1: RUP: Esfuerzo dedicado en cada iteración a cada workflow.

RUP gestiona los procesos de entrega de documentos y la autoría de los procesos. Además incluye plantillas, seguimiento de avances mediante hitos, informes, mentores de herramientas y un conjunto de roles asignables a cada participante.

En RUP también se indica el uso adecuado de los distintos diagramas UML:

Flujo	Diagrama UML
Modelado del Negocio	Casos de Uso Objetos
Requisitos	Casos de Uso + Estados
Análisis y Diseño	Casos de Uso → Clases Casos de Uso → Colaboración Organización Clases + Estados Secuencia
Implementación	CUALQUIERA
Pruebas	CUALQUIERA
Despliegue	Organización
Configuración, gestión,...	CUALQUIERA

Al ser iterativo, RUP promueve que se minimice el riesgo de obtención de un mal producto (o un producto no deseado) porque el sistema puede validarse con el cliente en cada iteración. De esta forma se potencia la robustez del producto y se incluye un marco en el que el problema de tener que gestionar requisitos incompletos, que es bastante frecuente, sea fácil de llevar.

La realización de las cuatro fases de RUP produce una

Ingeniería del Software Orientada a Objetos

generación del producto. Cada fase tiene una o más iteraciones de todos los flujos y finaliza en un **Hito**. Al finalizar cada fase, en estos hitos ha de cumplirse que:

- Fase de **Inicio**: se comprende el problema y se determina su ámbito.
- Fase de **Elaboración**: se capturan los requisitos y se produce un prototipo.
- Fase de **Construcción**: se realiza el diseño e implementación, terminando una versión Beta del producto.
- Fase de **Transición**: se suministra el producto final.

RUP es suficientemente genérico para simular fácilmente cualquiera de los ciclos de vida clásicos. Se puede configurar cambiando el número de iteraciones, el ciclo de vida empleado y seleccionando qué esfuerzo queremos dedicar en cada actividad.

También es capaz de englobar el propio proceso de la Programación eXtrema que se muestra a continuación, aunque algunas de sus prácticas (como la refactorización y la descentralización de la autoría de actividades) no están muy bien integradas.

4. Programación eXtrema: XP

La XP parte de un punto de vista que da una buena idea del porqué de su llamativo nombre:

Si los proyectos desarrollados con un diseño exhaustivo siguen teniendo errores y siguen retrasándose, ¿por qué perder tiempo en el diseño?

Aquí la actividad clave es la implementación. La XP se basa en la existencia de equipos pequeños de desarrolladores para los que es factible mantener una estrecha relación con el cliente. Esto les permite realizar reuniones con frecuencia y de esta forma se capturan requisitos sin retrasos y se implementan cuanto antes.

Las prácticas más comunes de la XP son:

- Programación en pares. Cada unidad de trabajo está formada por dos desarrolladores. Esto propicia la transmisión de conocimiento, distribución de las tareas y mantiene más constante el ritmo de trabajo de la unidad.
- Semanas de 40 horas. El tiempo de trabajo semanal de cada miembro del proyecto tiene que estar limitado y conviene que sea equitativo. Si no se adopta esta práctica, la realización de estimaciones temporales precisas se hace impracticable.
- Hay que implementar los tests antes de que la solución esté implementada.
- Se promueve el uso de refactorización y que se minimicen los comentarios en el código. La refactorización intenta compensar el esfuerzo del diseño no realizado. El mantenimiento de los comentarios tiene un coste alto y, en la mayoría de ocasiones, el código debería ser lo suficientemente autoexplicativo como para poder evitarlos sin riesgo de perder claridad. Esta medida hace énfasis en el hecho de que la vía principal de comunicación en el proyecto es el propio código.
- Integración continua. Hay que integrar los subsistemas tras cada modificación de forma que el producto esté ensamblado cuanto antes. Esta práctica también incluye la realización de los tests tras cada cambio.

Discontinuidad: ¿Sí o No?

Como mencionaba en la introducción, es sumamente importante que no se pierda información en los cambios de fase, porque en ese caso la conformidad del producto con respecto a los requisitos del cliente se vería afectada peligrosamente.

Una de las características más deseables de la Orientación a Objetos es que aporta continuidad entre el diseño y la implementación. Gracias a esto, los requisitos son traducibles a código de forma rápida y trazable. Sin embargo, si no se toman algunas medidas, la realización de un diseño de forma imprecisa causa la reaparición de esta discontinuidad.

Para evitar la discontinuidad, un primer paso puede ser completar los conocimientos de UML con OCL (Object Constraint Language). OCL es una extensión de UML que permite representar restricciones o condiciones invariantes que deben cumplirse. Por ejemplo:

```
context Compañía inv:
    self.empleado→notEmpty()
```

Esta restricción indica que toda compañía debe tener empleados y, por ello, fuerza a que algunos de ellos existan antes que la propia compañía. Las expresiones OCL pueden asociarse a cualquier elemento o relación de elementos de cualquier diagrama UML. En el ejemplo, el contexto indica el elemento asociado, la clase Compañía. El propósito de OCL es aportar formalidad de manera clara y legible. Es una herramienta con la que es posible evitar largos párrafos de lenguaje coloquial que podrían ser confusos.

Además de OCL, para conseguir un diseño preciso es aconsejable utilizar técnicas que indaguen en la experiencia de desarrollos previos y que utilicen amplias comunidades de desarrolladores. En esta línea de actuación se puede hacer uso de los Patrones (de forma preventiva) y la Refactorización (de forma correctiva).

1. Los Patrones

Los patrones encapsulan conceptos completos de programa que tienen traducción directa a código. Entre los patrones más conocidos se encuentran: el *Singleton*, el *Proxy* (control de accesos), el *Estado*, la *Máquina de Estados* (sustitución de implementaciones), los *Iteradores Seguros*, la *Factoría* (creación de objetos), el *Objeto Función*, el *Adaptador*, la *Fachada*, el *Observador* (llamadas de retorno o "callbacks"), etc.

Veamos como ejemplo clásico el *Singleton*, cuyo código

```
final class Singleton {
    private static Singleton s =
        new Singleton(5);
    private int attr;
    private Singleton(int param) { attr = param; }
    public static Singleton getReference() { return s; }
    public int getAttribute() { return attr; }
    public int setAttribute(int value) { attr = value; }
}
```

Ingeniería del Software Orientada a Objetos

Java asociado aparece en la tabla adjunta. El Singleton controla que solo exista una única instancia de una clase (se observará que es fácilmente convertible en un *pool* de objetos).

La clase es final para que no existan descendientes que incumplan las características del Singleton. El atributo que guarda la referencia al Singleton es estático para asegurar su existencia y privado para que esta no pueda cambiarse. El constructor es privado para que nadie más que el propio Singleton sea capaz de crear un objeto.

O sea que, básicamente, un patrón es una receta con un conjunto de características que, en la mayoría de casos, se pueden deducir con un poco de sentido común. Sin embargo, el uso de la receta nos asegura que el código está siendo usado por multitud de desarrolladores, con lo que esto conlleva acerca de su validez y posibilidades de reutilización.

2. La Refactorización

Refactorizar significa mejorar (eliminar redundancias que afecten al coste temporal y espacial) el diseño del código sin cambiar el comportamiento del sistema.

La refactorización es una tarea correctiva porque, al contrario que lo que ocurre con los patrones, se realiza cuando el código ya existe.

En Martin Fowler et al. (1999) aparecía una serie de trucos para mejorar el código:

- Convertir las asociaciones bidireccionales en unidireccionales cuando finalmente solo se utilice uno de los sentidos de la relación.
- Colapsar la jerarquía de herencia cuando no aporte lo suficiente para tener más de una entidad separada.
- Encapsular atributos tras operaciones *get()* y *set()*.
- Conviene esconder (como privados) los métodos de la interfaz que no se invoquen desde otros objetos.
- Encapsular la conversión repetida de tipos descendientes.
- Los nombres de métodos y atributos siempre han de ser autoexplicativos.

Como se observa a raíz de estos ejemplos, la refactorización es una tarea casi mecánica. Por ello, algunos editores y entornos de desarrollo (como JEdit o Eclipse) incluyen *plugins* de refactorización automática.

3. Desarrollo Software Orientado a Aspectos

Además de los anteriores factores, existen otras muchas consideraciones que deben realizarse durante el diseño. El AOSD o AOD (Desarrollo Orientado a Aspectos), es una aproximación al diseño de la arquitectura del sistema. En lugar de solo identificar requisitos como en la mayoría de aproximaciones, en el AOSD se diferencia entre asuntos, aspectos y requisitos.

Un *requisito* es una característica que debe estar presente en el sistema (los requisitos son propios de cada proyecto). Un *asunto* es una cuestión de interés (como lo son la seguridad, la persistencia, el rendimiento,...). Y un

aspecto es un conjunto de requisitos que está diseminado de tal forma que no pueden encapsularse claramente bajo ninguna unidad de código (paquetes, métodos o clases).

Además de centrarse en estos, la AOSD incluye un lenguaje de descripción de aspectos. Esta técnica se aplica principalmente en seguridad, sistemas operativos, planificadores en tiempo real y testing.

La primera de las áreas, la seguridad (en el sentido amplio de la palabra: seguridad de datos y accesos, mantenibilidad, independencia frente a errores de actualización), es el ejemplo más claro de la utilidad de

Algunas direcciones que pueden serle de ayuda:

Object Management Group: www.omg.org

Object Mentor: www.objectmentor.com

Objects by Design: www.objectsbydesign.com

Workflow Management Coalition:

www.wfmc.org

Rational: www.rationaledge.com

XP: www.extremeprogramming.org

los aspectos. Debido a la magnitud de los sistemas y a su división en subsistemas, hay que mantener muchas de las piezas que forman un sistema para asegurar la seguridad de este. Si se reúnen todas esas piezas bajo el concepto de un aspecto, se facilita el mantenimiento de la seguridad.

En Definitiva...

He aquí una serie de metodologías y técnicas que complementan a la Orientación a Objetos en el proceso de desarrollo de proyectos software.

El lector inquieto argumentará que se están obviando factores nada despreciables tales como herramientas de desarrollo integrado, plazos, lenguaje de programación, generadores de diagramas y esqueletos, estándares de calidad, documentación, control de cambios, asignación de recursos, etc.

Su ausencia en este artículo es una mera cuestión de espacio y ninguno de ellos debe desestimarse.

En definitiva, como cualquier proceso de ingeniería que se precie, la Ingeniería del Software persigue la resolución de problemas bastante complejos. La utilidad de las metodologías y ciclos de vida no se limita a conseguir que el producto se termine y funcione. Sus ventajas, que se han ido enumerando, son bastante intangibles y puede que tanto clientes como desarrolladores no las aprecien a primera vista. Sin embargo, el conocimiento y el uso adecuado de estas técnicas puede facilitar el trabajo crítico y evitar muchos errores de difícil detección y corrección.

Autor: Paco Castro

Más información: sidi@iti.upv.es