Introducción a la ingeniería del software

Jordi Pradel Miquel
Jose Raya Martos

PID_00230163

UOC
Universitat Oberta de Catalunya

Índice

Introducción

Este módulo debe servir para introducir a los futuros ingenieros de software en su disciplina. Por ello, lo que queremos es dar una visión general de las diferentes partes que forman la ingeniería del software, situarlas en contexto y relacionar unas con otras.

Más adelante, cuando se estudien los detalles de cada una de las partes (en estos mismos materiales o en futuras asignaturas), podremos volver atrás a este módulo y ver dónde encaja lo que se está estudiando dentro del contexto de la ingeniería del software.

Hemos tenido en cuenta una visión bastante amplia del ámbito de la ingeniería del software, no sólo desde el punto de vista estrictamente técnico, sino también teniendo en cuenta los aspectos organizativos, dado que es normal que del ingeniero de software se espere que sepa organizar el desarrollo y el mantenimiento del software.

Empezaremos estudiando la ingeniería del software como disciplina de la ingeniería. Es importante que el ingeniero de software sepa cuáles son las particularidades de su ingeniería y también cuáles son las similitudes con las otras ingenierías y que sea capaz de establecer paralelismos y ver las diferencias. También veremos cómo se llegó a la conclusión de que, pese a las diferencias, la ingeniería era el enfoque más adecuado para el desarrollo de software.

Desde el punto de vista organizativo, veremos cuáles son las actividades que se deben llevar a cabo para desarrollar un producto de software y de qué manera las podemos organizar en función de las características del producto por desarrollar.

En concreto, veremos tres maneras diferentes de organizar un proyecto de desarrollo:

1) siguiendo el ciclo de vida en cascada (Métrica 3),

2) siguiendo el ciclo de vida iterativo e incremental (Open UP) y

3) siguiendo los principios ágiles (Scrum).

Desde el punto de vista técnico, veremos (sin estudiarlo con detalle) algunas de las herramientas y técnicas que se usan hoy en día dentro de la profesión: la orientación a objetos, los bastimentos, el desarrollo basado en componentes, el desarrollo orientado a servicios, los patrones, las líneas de producto, la arquitectura dirigida por modelos y los lenguajes específicos del dominio.

Finalmente, un ingeniero de software debe conocer los estándares establecidos en su profesión. Por ello, veremos algunos estándares relacionados con los contenidos del módulo: UML, SWEBOK, CMMI y PMBOK.

Objetivos

Los objetivos que el estudiante debe haber alcanzado una vez trabajados los contenidos de este módulo son los siguientes:

1. Entender qué es la ingeniería del software y situarla en contexto.

2. Entender las peculiaridades de la ingeniería del software comparada con otras ingenierías.

3. Saber identificar los diferentes roles y actividades que participan en un proyecto de desarrollo de software.

4. Conocer algunos de los métodos de desarrollo más utilizados.

5. Conocer algunas de las técnicas propias de la ingeniería del software.

6. Conocer los principales estándares de la ingeniería del software.

1. ¿Qué es la ingeniería del software?

Antes de empezar con el estudio de la ingeniería del software, debemos ponernos en contexto y definir qué es el software, qué actividades relacionadas con el software cubre la ingeniería del software y cómo hemos llegado, como industria, a la conclusión de que la ingeniería es el enfoque adecuado para estas actividades.

1.1. Software y hardware

Denominamos *software* a todo aquello intangible (no físico) que hay en un ordenador, incluyendo el conjunto de programas informáticos que indican la secuencia de instrucciones que un ordenador debe ejecutar durante su funcionamiento (también denominado *código*) y el resto de los datos que este ordenador manipula y almacena.

Código y software

En general, salvo que indiquemos lo contrario, en esta asignatura abusaremos del lenguaje al usar la palabra software para referirnos al código (y excluiremos, por lo tanto, los datos).

> Más formalmente, el IEEE define software como: "El conjunto de los programas de computación, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de cómputo".

Referencia bibliográfica

IEEE Std (1993). *IEEE Software Engineering Standard: glossary of Software Engineering Terminology*. IEEE Computer Society Press.

Por oposición, denominamos *hardware* al conjunto de componentes físicos de un ordenador. Este hardware ofrece una serie de instrucciones que el ordenador es capaz de ejecutar cuando ejecuta un programa.

1.2. El desarrollo de software

Como acabamos de ver, para programar un ordenador hay que escribir una secuencia de instrucciones entre las que el ordenador ofrece. El software tiene una forma ejecutable, que es la lista de instrucciones en un formato que el ordenador es capaz de entender y ejecutar directamente, el código máquina.

Pero el código máquina no es legible para las personas. Además, el juego de instrucciones posibles ofrecido por el hardware de un ordenador es relativamente reducido y sencillo, lo que provoca que sea bastante complicado escribir programas complejos usando directamente este juego de instrucciones. Por esta razón, para programar los ordenadores se usan lenguajes de programación y bibliotecas de software, que permiten escribir las instrucciones a un nivel de abstracción más elevado, usando instrucciones compuestas, más complejas.

Llamamos **código fuente** a la manera legible por las personas en la que se escribe el software.

Ejemplo de comparación de dos números

Supongamos que nuestro ordenador no permite comparar dos números directamente y que sólo permite restarlos y saltar a una instrucción concreta en función de si el resultado era más grande, menor o igual a 0.

Si queremos comparar dos números (*num1* y *num2*) entre sí para saber si son iguales, el código máquina debe mover los números a uno de los registros de la CPU (operación MOV), restarlos (operación CMP) y saltar a la etiqueta que corresponda. Por lo tanto, cada vez que el programador quiera comparar *a* y *b* deberá escribir dos instrucciones: la resta y la comparación.

En cambio, los lenguajes de programación permiten escribir una única instrucción *if*(*num1>num2*) y hacer la traducción a la serie de instrucciones de código máquina de manera automática. De este modo, el código fuente es más fácil de escribir y de leer.

En lenguaje ensamblador x86:

```
 mov ax, num1
 mov bx, num2
 cmp ax, bx
 jg num2MasGrande
 ;num1>num2
 jmp final
num2MasGrande:
  ;num2>=num1
final:
```

En lenguaje C (y derivados):

```
if (num1>num2) {
  //num1 > num2
} else {
  //num2 >= num1
}
```

Normalmente, el software se desarrolla con el objetivo de cubrir las necesidades de un cliente u organización concreta, de satisfacer las necesidades de un determinado grupo de usuarios (y venderles o darles el software para que lo usen) o para uso personal.

El desarrollo de software es el acto de producir o crear software.

A pesar de que el desarrollo de software incluye la programación (la creación del código fuente), usamos el término *desarrollo de software* de una manera más amplia para referirnos al conjunto de actividades que nos llevan desde una determinada idea sobre lo que queremos hasta el resultado final del software.

Entre estas actividades, podemos encontrar ejemplos como la compilación, el estudio y la documentación de las necesidades de los usuarios, el mantenimiento del software una vez se empieza a usar, la coordinación del trabajo en equipo de las diferentes personas que intervienen en el desarrollo, la redacción de manuales y ayudas de uso para los usuarios, etc.

Como ha sucedido en otras áreas, cuando la calidad obtenida y el coste del desarrollo son importantes, las organizaciones y empresas que desarrollan software convierten estas actividades en una ingeniería.

A pesar de que todavía hay mucho debate sobre si el desarrollo de software es un arte, una artesanía o una disciplina de ingeniería, lo cierto es que desarrollar software de calidad con el mínimo coste posible ha resultado ser una actividad, en general, muy compleja. En estos materiales, estudiaremos el desarrollo de software como ingeniería y no tendremos en cuenta los otros enfoques posibles.

1.3. El ámbito de la ingeniería del software

Los primeros computadores electrónicos y los primeros programas informáticos estaban orientados a la realización de cálculos matemáticos (de ahí el término *computadora*), pero hoy en día podemos encontrar software prácticamente en todas partes, desde los sistemas de información de cualquier organización hasta un reloj de pulsera, una motocicleta o las redes sociales en Internet.

Las enormes diferencias entre los diferentes tipos de software provocan que la manera de desarrollar unos y otros sea totalmente distinta. Así, una red social en Internet puede actualizar la aplicación que usan sus usuarios con relativa facilidad (sólo necesita actualizar sus servidores), mientras que actualizar el software que controla la centralita electrónica de todos los coches de un determinado modelo puede tener un coste enorme para el fabricante.

Por lo tanto, una de las primeras tareas del ingeniero de software es situar el ámbito o área en la que se aplicará el software que se ha de desarrollar. En este sentido, podemos realizar una clasificación de las áreas potenciales de aplicación de la ingeniería del software basada en la que ofrece Roger Pressman (2005):

- **Software de sistemas**. Son programas escritos para dar servicio a otros programas, como los sistemas operativos o los compiladores. Este tipo de programas suelen interactuar directamente con el hardware, de manera que sus usuarios no son los usuarios finales que usan el ordenador, sino otros programadores.

- **Software de aplicación**. Son programas independientes que resuelven una necesidad específica, normalmente de una organización, como por

ejemplo el software de gestión de ventas de una organización concreta. Pueden ser desarrollados a medida (para un único cliente) o como software de propósito general (se intentan cubrir las necesidades de varios clientes y es habitual que éstos utilicen sólo un subconjunto de la funcionalidad total).

- **Software científico y de ingeniería**. Muy enfocados al cálculo y a la simulación, se caracterizan por la utilización de algoritmos y modelos matemáticos complejos.

- **Software empotrado**. Es el software que forma parte de un aparato, desde el control de un horno hasta el ordenador de a bordo de un automóvil. Se caracteriza por las limitaciones en cuanto a recursos computacionales y por estar muy adaptado al producto concreto que controla.

- **Software de líneas de productos**. Es software diseñado para proporcionar una capacidad específica pero orientado a una gran variedad de clientes. Puede estar enfocado a un mercado muy limitado (como la gestión de inventarios) o muy amplio (como una hoja de cálculo).

- **Aplicaciones web**. Las aplicaciones web, independientemente de que sean un paquete o a medida, tienen una serie de características que las hacen diferentes del resto del software. Se caracterizan por unificar fuentes de datos y diferentes servicios en entornos altamente distribuidos.

- **Software de inteligencia artificial**. Estos programas usan técnicas, herramientas y algoritmos muy diferentes del resto de los sistemas y, por lo tanto, tienen una problemática propia. Pertenecen a esta categoría los sistemas expertos, las redes neuronales y el software de reconocimiento del habla.

Estas categorías no son necesariamente excluyentes, por lo que nos podemos encontrar con un software de aplicación desarrollado como una aplicación web o un software empotrado desarrollado como línea de productos. Lo cierto es que cada categoría de las mencionadas tiene su problemática específica.

Dado que no podemos estudiar todas estas problemáticas, en estos materiales nos centraremos en un tipo concreto de software: el software de aplicación desarrollado a medida, concretamente, el software para sistemas de información.

Un sistema de información es cualquier combinación de tecnología de la información y actividades humanas que utilizan esta tecnología para dar apoyo a la operación, gestión o toma de decisiones.

Sistema de información y sistema informático

No debemos confundir un sistema de información con un sistema informático. Un sistema de información puede estar formado por ninguno, uno o más sistemas informáticos, así como por personas y otros soportes de información.

Por ejemplo, el sistema de información de gestión de la logística de una empresa puede estar formado por un sistema informático de gestión de pedidos, un albarán en papel y varias personas. Por otro lado, este albarán en papel se podría sustituir por otro sistema informático o por el mismo sistema informático de gestión de pedidos.

El software para sistemas de información es un tipo de software que gestiona una cierta información mediante un sistema gestor de bases de datos y soporta una serie de actividades humanas dentro del contexto de un sistema de información.

1.4. ¿Qué es la ingeniería del software?

El IEEE define la ingeniería como "la aplicación de un enfoque sistemático, disciplinado y cuantificable a las estructuras, máquinas, productos, sistemas o procesos para obtener un resultado esperado" y, más concretamente, la ingeniería del software como "(1) La aplicación de un enfoque sistemático, disciplinado y cuantificable en el desarrollo, la operación y el mantenimiento del software; es decir, la aplicación de la ingeniería al software. (2) El estudio de enfoques como en (1)".

Referencia bibliográfica

IEEE Std (1993). *IEEE Software Engineering Standard: glossary of Software Engineering Terminology*. IEEE Computer Society Press.

El **desarrollo** es, como hemos dicho anteriormente, el proceso que lleva a la producción o creación del producto de software; **la operación** consiste en ejecutar el producto de software dentro de su entorno de ejecución para llevar a cabo su función; finalmente, el **mantenimiento** comprende la modificación posterior del producto de software para corregir los errores o adaptarlo a nuevas necesidades.

Por lo tanto, cuando hablamos de ingeniería del software no solamente estamos hablando de una manera de desarrollar software, sino que también debemos tener en cuenta la vida posterior del producto una vez creado. La ingeniería del software consiste en llevar a cabo todas estas actividades de manera que podamos medir, cuantificar y analizar los diferentes procesos relacionados con la vida del producto de software.

Este enfoque nos permite extraer conclusiones aplicables a futuras actividades relacionadas con el producto de software y responder a preguntas como qué recursos son necesarios para desarrollar un nuevo producto, cuánto tiempo será necesario para añadir una nueva funcionalidad a un producto ya existente o qué riesgos podemos encontrar.

Por lo tanto, todo esto es muy difícil de conseguir sin un enfoque sistemático y cuantificable y acerca la ingeniería del software a las otras ingenierías, mientras que la aleja de la creación artística.

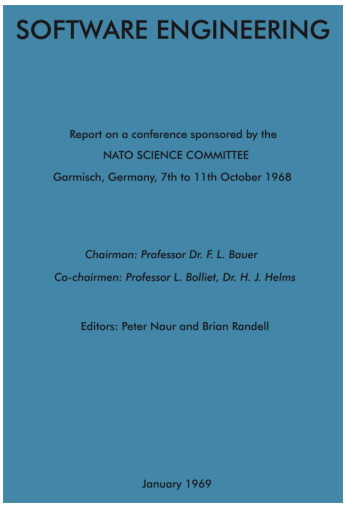
1.5. Historia de la ingeniería del software

Originalmente, el software era un producto gratuito que se incluía al comprar hardware y era desarrollado, principalmente, por las compañías fabricantes del hardware. Unas pocas compañías desarrollaban software a medida, pero no existía el concepto de software empaquetado, como producto.

El desarrollo de software no se gestionaba según una planificación y era prácticamente imposible predecir los costes y el tiempo de desarrollo. Pero ya se aplicaban algunas técnicas de reutilización, como la programación modular.

El término *ingeniería del software* se empezó a usar hacia finales de los años cincuenta, pero el punto de inflexión que lo convirtió en un término usado globalmente fue una conferencia del comité científico de la OTAN, en octubre de 1968, que definió formalmente el término *ingeniería del software*.

La conferencia de la OTAN se llevó a cabo porque el software estaba adquiriendo cada vez más importancia económica y social. Sin embargo, en la misma conferencia se detectó que el desarrollo de software estaba muy lejos de lograr los niveles de calidad, productividad y coste previstos, hecho que se denominó la *crisis del software*. La crisis consistía en la dificultad de escribir software correcto, entendible y verificable, lo que causaba que los proyectos tuvieran costes mucho más elevados de lo previsto, que no se acabaran en los plazos esperados, que tuvieran una calidad baja, que no cumplieran los requisitos, etc.



La primera conferencia sobre ingeniería del software se organizó en la ciudad de Garmish, Alemania, entre los días 7 y 11 de octubre de 1968, y fue esponsorizada por la OTAN. Las actas de la conferencia se publicaron en enero de 1969 y se pueden encontrar en el enlace siguiente: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

Así, por ejemplo, el informe Chaos, elaborado por Standish Group en 1995, elaboró un estudio sobre el desarrollo de software en Estados Unidos y detectó que el 90% de los proyectos estudiados no cumplían los objetivos de tiempos, coste o calidad. Algunas cifras alarmantes de este mismo informe mostraban que el 31% de los proyectos de software eran cancelados antes de completarse y que sólo el 16% se concluían en el tiempo, presupuesto y alcance planificados.

Informe Chaos

El informe Chaos ha recibido críticas respecto a su negatividad, dado que considera exitosos sólo los proyectos en los que se cumplen los tres requisitos (plazo, presupuesto y alcance).

Independientemente de lo que consideremos como proyecto exitoso, hay cifras que son indiscutiblemente significativas, como el hecho de que se cancelase casi un tercio de los proyectos.

Durante décadas, empresas e investigadores se centraron en superar la crisis del software y desarrollaron nuevas herramientas, tecnologías y prácticas, buscando infructuosamente solucionar de manera definitiva todos los problemas existentes; en el mejor de los casos, se fueron introduciendo paulatinamente mejoras incrementales.

El 1986, Fred Brooks publicó un artículo (Brooks, 1987), titulado "No Silver Bullet" (que podríamos traducir como "No hay soluciones mágicas"), en el que argumentaba que no había una solución única para el problema. Ningún desarrollo en tecnología o gestión, decía, introducirá mejoras ni siquiera de un orden de magnitud en productividad, fiabilidad o simplicidad, al menos no durante una década. También argumentaba que no podemos esperar mejoras de dos órdenes de magnitud cada dos años como las que se producían en el hardware.

"No Silver Bullet"

El título del artículo de Fred Brooks, "No Silver Bullet", literalmente "No hay balas de plata", utiliza una metáfora en la que la solución definitiva mencionada es una bala de plata para matar al monstruo de los problemas de la ingeniería del software, que es el hombre lobo de las leyendas.

La causa de estas afirmaciones, según Brooks, era que las mejoras que se estaban introduciendo podían paliar la complejidad accidental, pero no la complejidad esencial, inherente al desarrollo de software. Así, las soluciones ya aplicadas entonces –como los lenguajes de programación de alto nivel o las herramientas integradas de desarrollo de software– y las soluciones que se consideraban de futuro –como la programación orientada a objetos, la inteligencia artificial o el prototipado– solucionan complejidades accidentales y, a pesar de que introducen mejoras, no lo llegan a hacer en órdenes de magnitud.

En los años noventa, el nacimiento y el crecimiento exponencial de Internet provocaron un crecimiento muy rápido de la demanda de software especializado, sobre todo en la Red. Esta demanda creciente, y el hecho de que muchas organizaciones pequeñas requirieran también desarrollo de software, introdujo la necesidad de soluciones de software más simples, rápidas de desarrollar y baratas. Así, a pesar de que los grandes proyectos de software continuaban usando las metodologías desarrolladas anteriormente, en los proyectos más pequeños se aplicaban metodologías más ligeras.

A comienzos de la década del 2010, se continúan buscando y aplicando soluciones para mejorar la ingeniería del software. El informe Chaos del 2009 muestra que los proyectos considerados 100% exitosos han pasado del 16% en 1994 al 32% en el 2009, mientras que los cancelados han bajado del 31 al 24%. Estas cifras, a pesar de haber mejorado, se consideran todavía bastante negativas.

La gran mayoría de los ingenieros está de acuerdo con Brooks en que no existen soluciones mágicas y buscan soluciones incrementales que vayan mejorando los resultados obtenidos en la ingeniería del software. Algunas de estas soluciones son las líneas de producto, el desarrollo guiado por modelos, los patrones o las metodologías ágiles de desarrollo. De éstas y otras soluciones hablaremos a lo largo de estos materiales.

1.6. La ingeniería del software comparada con las otras ingenierías

Una de las ventajas de aplicar la ingeniería a las actividades relacionadas con el software es que nos permite aprovechar el conocimiento generado en otras disciplinas de la ingeniería para mejorar el modo como gestionamos estas actividades.

Así, a lo largo del tiempo, se han ido aplicando diferentes metáforas (con más o menos éxito) para extrapolar el conocimiento adquirido en los diferentes ámbitos de la ingeniería al mundo del software y los sistemas de información.

La metáfora de la construcción

Un caso muy común es el de la construcción. El arquitecto crea unos planos que deben estar terminados antes de empezar la construcción del edificio, los albañiles son fácilmente intercambiables, dado que las instrucciones que han de seguir son muy claras, y una vez finalizado el trabajo el cliente se puede hacer cargo del mantenimiento del edificio sin necesidad de contactar con los albañiles que lo construyeron.

Sin embargo, al aplicar esta metáfora a menudo no se tienen en cuenta factores como que el edificio se construye en el lugar en el que se usa, mientras que el software se desarrolla en un entorno diferente de aquel en el que se usa; o que el software de calidad se debe poder utilizar en entornos diferentes; o que el coste de hacer copias del software es casi nulo en comparación con el coste de diseñarlo.

Uno de los peligros de las metáforas es que, a menudo, se aplican de acuerdo con un conocimiento incompleto del ámbito de partida, lo que provocará errores a la hora de trasladar el conocimiento y las prácticas de una ingeniería a otra.

Construcción sin planos

En el ejemplo anterior hemos supuesto que el edificio está totalmente diseñado antes de empezar la construcción y que, por lo tanto, no hay que modificar el diseño, pero en la práctica muchas veces los constructores se encuentran con problemas no previstos que los obligan a modificar el diseño original.

Por ejemplo, el templo de la Sagrada Familia de Barcelona se empezó a construir en 1882, pero Gaudí lo replanteó totalmente en 1883. El arquitecto modificó la idea original a medida que avanzaba la construcción, hasta que murió en 1926, sin dejar planos ni directrices sobre cómo había que continuar la obra, que se continuó construyendo durante todo el siglo XX y parte del XXI. Ésta es una historia muy diferente de la que tenemos en mente cuando hablamos de cómo funciona la construcción.

Por lo tanto, es muy importante conocer bien las características inherentes al software que lo diferencian de los demás productos industriales para poder aplicar con éxito el conocimiento generado en otras ingenierías. Podríamos destacar las siguientes:

1) El software es intangible. El software es un producto intangible cuya producción no consume ninguna materia prima física: podríamos decir que la materia prima del software es el conocimiento. Como consecuencia, la gestión de los procesos relacionados con su desarrollo y mantenimiento es diferente de la de muchos productos industriales.

2) El software no se manufactura. Como sucede con cualquier producto digital, una vez desarrollado el software, crear copias tiene un coste muy bajo y las copias creadas son idénticas a la original. Así, no hay un proceso de manufactura de la copia y, por lo tanto, el conocimiento derivado de la manufactura de productos industriales no es trasladable al software. En cambio, el conocimiento relacionado con el desarrollo del producto sí que lo será.

3) El software no se desgasta. A diferencia de los productos tangibles, el software no se desgasta, lo que hace que su mantenimiento sea bastante diferente del mantenimiento de un producto industrial tangible. Por ejemplo, si un software falla, como todas las copias son idénticas al original, todas tendrán el mismo error; no hay piezas de repuesto para cambiar. Por este motivo, deberemos revisar todo el proceso de desarrollo del software para detectar en qué punto se introdujo el error y corregirlo.

4) El software queda obsoleto rápidamente. El rápido cambio tecnológico genera que el software quede obsoleto con relativa rapidez. Esto incrementa la presión para conseguir desarrollarlo de manera rápida y con poco coste, dado que su ciclo de vida es muy corto si lo comparamos con otros productos industriales, como los coches, los aviones o las carreteras.

Otro rasgo distintivo de la ingeniería del software es que es una industria relativamente nueva. Como se ha comentado, la primera mención reconocida del término *ingeniería del software* fue en 1968 y los primeros ordenadores electrónicos aparecieron en la década de los cuarenta. Por lo tanto, la experiencia acumulada es relativamente poca si la comparamos con otras ingenierías. Este hecho se ve agravado por la rápida evolución de la tecnología, fundamentalmente en dos ejes:

- Por un lado, las nuevas posibilidades tecnológicas hacen que cambie totalmente la naturaleza de los productos que estamos desarrollando; por ejemplo, la aparición de las interfaces gráficas en los años ochenta, la consolidación del acceso a Internet en los años noventa o la popularización del acceso a Internet desde el móvil en la primera década del siglo XXI

dieron lugar a nuevos tipos de productos y a nuevas necesidades prácticamente inimaginables diez años atrás.

- Por otro lado, el aumento de la potencia de cálculo de los ordenadores también ha provocado cambios fundamentales en las herramientas que se usan para desarrollar software, tanto a escala de lenguajes y paradigmas de programación (estructurados, orientados a objetos, dinámicos, etc.), como de las propias herramientas (entornos de desarrollo, herramientas CASE, etc.).

Herramientas CASE

Las herramientas CASE (*computer aided software engineering*) son las herramientas que nos ayudan a llevar a cabo las diferentes actividades de la ingeniería del software, como la creación de modelos, la generación de documentación, etc.

Esto provoca que gran parte del conocimiento acumulado a lo largo de los años haya quedado obsoleto debido a las importantes diferencias entre el contexto en el que este conocimiento se generó y el nuevo contexto en el que se debe aplicar.

Finalmente, no podemos dejar de mencionar otro rasgo distintivo de la ingeniería del software, como es la aparición del software libre. El software libre ha transformado enormemente la manera de desarrollar y mantener cualquier tipo de software y ha promovido la creación de estándares de facto, la reutilización de software y la difusión del conocimiento respecto a como ha sido desarrollado. Este nivel de transparencia y colaboración no se encuentra, a estas alturas, en ninguna otra ingeniería.

2. Organización de la ingeniería del software

Hemos dicho anteriormente que la ingeniería del software implica ser sistemáticos en el desarrollo, la operación y el mantenimiento del software, para lo que es necesario definir cuál es el proceso que hay que seguir para llevar a cabo estas tareas; ésta es la finalidad de los métodos de desarrollo.

> El IEEE define un método diciendo que "describe las características del proceso o procedimiento disciplinado utilizado en la ingeniería de un producto o en la prestación de un servicio".

Por lo tanto, para aplicar ingeniería a las actividades relacionadas con el software lo debemos hacer por medio de la aplicación de un método. Este método puede ser aplicable al desarrollo, a la operación o al mantenimiento del software o a cualquier combinación de los tres.

2.1. Organización del desarrollo, operación y mantenimiento

Existen diferencias importantes sobre cómo se deben organizar las diferentes actividades relativas al desarrollo, la operación y el mantenimiento del software.

Las actividades de operación suelen ser continuas, mientras que el desarrollo suele ser temporal: el desarrollo tiene un inicio claro (el momento en el que se plantea la posibilidad de desarrollar el producto) y un final también bastante claro, sea con éxito o fracaso.

Otra diferencia importante es que las actividades de operación suelen ser repetitivas, mientras que el desarrollo proporciona un resultado único (dos procesos de desarrollo darán lugar a dos productos diferentes).

Por ello, a pesar de compartir ciertas similitudes, la organización del desarrollo es muy diferente de la organización de la operación del software.

Mientras que el desarrollo se suele organizar a modo de proyectos, la operación se organiza de acuerdo con servicios.

El mantenimiento se puede organizar de manera continua, especialmente en el caso del mantenimiento correctivo (que se efectúa para corregir errores en el software), o de manera temporal (a modo de proyecto), como en el caso del mantenimiento evolutivo (que se hace para modificar el producto de acuerdo con las nuevas necesidades detectadas con posterioridad a su desarrollo).

Debido a estas diferencias, a pesar de existir métodos que definen procesos para las tres actividades (desarrollo, operación y mantenimiento), muchos de ellos se centran en una única de éstas. Así, por ejemplo, hay muchos métodos de desarrollo que no definen el proceso que se debe seguir para operar el software.

En adelante, nos centraremos en el estudio de los proyectos de desarrollo de software y de mantenimiento.

Un proyecto de desarrollo de software tiene, como hemos dicho anteriormente, un inicio y un final muy determinados. Este final puede estar determinado por el éxito o fracaso del proyecto o por otras circunstancias, como la desaparición de la necesidad que hay que cubrir.

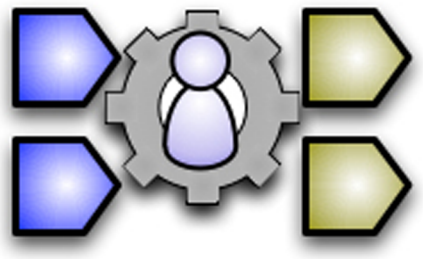
Lecturas recomendadas

Si queréis ampliar sobre la gestión de proyectos, podéis consultar la guía PMBOK publicada por el Project Management Institute (PMI) y, si os interesa la gestión de la operación del software, podéis consultar el estándar ITIL.

2.2. Organización de los proyectos de desarrollo

A lo largo del tiempo se han ido definiendo diferentes métodos para el desarrollo de software, muchos basados en los métodos aplicados en otras disciplinas de la ingeniería. Cada uno de estos métodos define uno o más procesos de desarrollo. La descripción de un proceso incluye, normalmente:

- Qué **tareas** y en qué orden se deben llevar a cabo.
- Qué **roles** deben tener las diferentes personas que participan en el desarrollo, cuál es la responsabilidad de cada rol y qué tareas deben llevar a cabo.



La descripción de un proceso indica los artefactos de entrada de una tarea, el rol responsable de llevarla a cabo y los artefactos de salida (que servirán de entrada para la tarea siguiente).

- Qué **artefactos** (documentos, programas, etc.) deben usarse como punto de partida para cada tarea y qué se debe generar como resultado.

Los procesos definidos por un método se basan en criterios tan variados como el alcance (hay métodos que sólo definen el proceso de desarrollo de software, mientras que otros tienen en cuenta otros aspectos, como el proceso que llevará a una organización a decidir si desarrolla el software o no), el tipo de organización al que va dirigido (pequeñas empresas, Administración pública, el Ejército, etc.) o el tipo de producto que se quiere generar (una aplicación de gestión de las vacaciones de los empleados de una empresa, el portal de Hacienda para presentar la declaración de la renta, etc.).

2.3. Actividades de la ingeniería del software

Tal y como hemos visto, los métodos definen qué tareas se llevarán a cabo en cada proceso. A pesar de que cada método tiene sus particularidades, es cierto que hay una serie de tareas que se deben llevar a cabo en todo proyecto de desarrollo con independencia de cómo se organice.

A lo largo de este módulo, usaremos el término *actividad* para referirnos a un conjunto de tareas relacionadas entre sí. Distinguimos actividades de tareas en el sentido de que un proceso define tareas concretas (generar el modelo conceptual, crear el diagrama de Gantt con la planificación del proyecto) que pueden diferir de un método a otro, mientras que la actividad (modelización, planificación) sería la misma.

Cada método puede dar más o menos importancia a cada una de las actividades que describimos a continuación, pero no las puede obviar si lo queremos considerar como un método completo de desarrollo de software.

2.3.1. Gestión del proyecto

La gestión de proyectos es una disciplina general, común para todo tipo de proyectos de las tecnologías de la información y comunicación (TIC) y otros ámbitos, como la construcción, la ingeniería y la gestión de empresas, entre otros, e incluye todos los procesos necesarios para la dirección, la gestión y la administración de cualquier proyecto, con independencia de qué producto concreto se esté construyendo. El objetivo principal es asegurar el éxito del proyecto.

En el caso de los proyectos de desarrollo de software, el método general de gestión de proyectos se tiene que complementar con los métodos, las técnicas y las herramientas propias de los proyectos de desarrollo de software.

Hay quien considera que la gestión del proyecto, al no ser una actividad específica del desarrollo de software, no forma parte del ámbito de la ingeniería del software. En cualquier caso, forma parte del proyecto de desarrollo y, por lo tanto, los métodos de desarrollo de software lo deben tener en cuenta.

A continuación, se indican algunas de las tareas relacionadas con la gestión del proyecto:

- **Estudio de viabilidad**. Antes de empezar propiamente el proyecto, habrá que detectar una necesidad en la organización que lo encarga y elaborar un estudio de alternativas y costes para decidir si el proyecto de desarrollo es o no la mejor opción para satisfacer las necesidades mencionadas. Para hacerlo, habrá que anticipar el coste del proyecto (qué recursos se deben invertir para el desarrollo) y cuánto tiempo será necesario para desarrollarlo.

- **Estimación**. Tal y como hemos dicho antes, hay que hacer una valoración del coste del proyecto, del tiempo necesario para llevarlo a cabo y de la relación entre recursos y tiempos: ¿cómo se modifica la estimación de tiempo si añadimos o quitamos recursos? Esta relación no suele ser lineal. El ejemplo clásico es que, mientras que una mujer puede gestar una criatura en nueve meses, nueve mujeres no la pueden gestar en un mes.

- **Definir claramente los objetivos del proyecto, que determinarán su éxito o fracaso**. Por ejemplo, un proyecto puede fracasar porque, una vez iniciado, se descubra que es imposible cumplir sus objetivos con los recursos o el tiempo disponible (y, por lo tanto, sea necesario cancelar el proyecto).

- **Formar el equipo de desarrollo teniendo en cuenta la estimación de recursos hecha inicialmente**. Este equipo puede estar formado por personas con dedicación completa al proyecto o con dedicación parcial. Cada vez más, es más habitual que estos equipos tengan, además de los desarrolladores, personas que representan a los *stakeholders*.

Stakeholder

El término inglés *stakeholder* hace referencia a cualquier persona u organización interesada, afectada o implicada en el funcionamiento del software que se desarrolla.

- **Establecer hitos** que nos permitan llevar a cabo las actividades de seguimiento y control del proyecto para verificar su buen funcionamiento.

- **Identificar riesgos que puedan poner en peligro el éxito del proyecto**. No sólo desde el punto de vista tecnológico (la obligación de utilizar tecnologías nuevas y poco probadas, etc.), sino desde el de todos (falta de apoyo por parte de la organización, problemas legales, etc.).

Una vez se ha decidido iniciar el proyecto y se ha efectuado la planificación inicial, la gestión del proyecto continúa, dado que debe observarse el progreso y compararlo con la previsión inicial para validar las suposiciones iniciales y hacer las correcciones oportunas en caso de estar equivocadas.

La gestión del proyecto está muy influenciada por el método de desarrollo empleado, ya que éste determinará las tareas que se han de llevar a cabo y también el orden en el que hacerlo. El método de desarrollo también indicará qué artefactos se generan durante la planificación y cómo se realiza el seguimiento y control del proyecto.

Nota

Respecto a la gestión de proyectos, no la trataremos en esta asignatura sino que hablaremos de ella ampliamente en una asignatura posterior.

Finalmente, podemos decir que, en última instancia, la gestión del proyecto consiste en equilibrar tres variables:

1) el alcance del proyecto (qué se debe incluir y qué no),

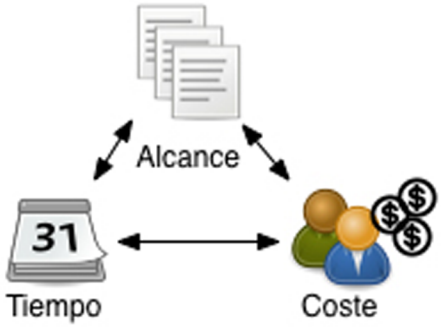
2) el tiempo (cuándo se finalizará la ejecución del proyecto) y

3) el coste (en recursos humanos y en recursos materiales).

Otras variables

Además del alcance, el tiempo y el coste, otros autores como Wysocki (2009) añade la calidad como variable y distingue entre el coste (presupuesto) y los recursos (principalmente, las personas).

Las tres variables de la gestión de proyectos: tiempo, alcance y coste



Un cambio en cualquiera de estas tres variables se debe compensar modificando, como mínimo, una de las otras dos (similar a lo que en teoría de juegos se denomina *juego de suma cero*).

Juego de suma cero

En teoría de juegos, un juego de suma cero describe una situación en la que la ganancia o la pérdida en la que incurre uno de los participantes se compensa exactamente por las pérdidas o ganancias del resto de los participantes. Así, si sumamos todas las pérdidas y todas las ganancias, el resultado final es 0. Por ejemplo, una partida de póquer sería un juego de suma cero, dado que el dinero que un jugador gana lo debe perder otro jugador (y viceversa).

2.3.2. Identificación y gestión de los requisitos

Esta actividad implica la comunicación y colaboración con los *stakeholders*, fundamentalmente para encontrar cuáles son los requisitos del producto que se debe desarrollar.

Ved también

Hablaremos extensivamente de la gestión de requisitos en el módulo "Requisitos".

Según la guía SWEBOK (2004), los requisitos "expresan las necesidades y restricciones que afectan a un producto de software que contribuye a la solución de un problema del mundo real".

Ved también

Podéis encontrar más información sobre SWEBOK en el subapartado 5.2 de este módulo didáctico.

Por lo tanto, los requisitos nos sirven para delimitar cuál de las posibles soluciones al problema son adecuadas (las que cumplen los requisitos) y cuáles no.

La identificación y gestión de requisitos está muy relacionada con el alcance del proyecto, dado que son los requisitos los que determinan este alcance. Las principales problemáticas que hay que vencer en esta tarea son las propias de cualquier actividad de comunicación:

- **Diferencias respecto a la información con la que trabajan las diferentes partes**. Los *stakeholders* tienen información sobre el producto que los desarrolladores no tienen, mientras que los desarrolladores tienen información sobre la tecnología que los *stakeholders* no poseen. Esto puede condicionar la visión del problema que tienen unos y otros y puede afectar negativamente a la transferencia de información.

- **Limitaciones del canal utilizado**. Cualquier canal de comunicación impone limitaciones. Por ejemplo, si la comunicación es escrita, se pierde el lenguaje no verbal, mientras que, si es verbal, se pierde la posibilidad de revisión que da la documentación escrita.

- **Limitaciones del lenguaje utilizado**. El lenguaje natural es propenso a la ambigüedad, razón por la que se han desarrollado los lenguajes formales de especificación; estos lenguajes, sin embargo, suelen ser menos expresivos que el lenguaje natural. Además, sólo sirven para la comunicación si todas las partes los entienden correctamente.

- **Dificultad de definir el mejor sistema posible**. Otro problema asociado a esta actividad es conseguir que los *stakeholders* comuniquen exactamente los requisitos del mejor sistema posible, dado que es habitual que, al describir el sistema que quieren, estén condicionados por el conocimiento que tienen sobre sistemas parecidos o, sencillamente, que no se les ocurran algunas posibilidades.

Para solucionar estos problemas, es muy habitual el uso de técnicas basadas en retroalimentación durante el análisis de requisitos (como el prototipado, que consiste en enseñar una versión con aspecto similar al producto final pero que no implementa la funcionalidad real) cuando esto es posible.

Otra técnica muy habitual es el uso de modelos, a pesar de que, en este caso, su utilidad está condicionada al hecho de que los usuarios/*stakeholders* entiendan la notación del modelo.

2.3.3. Modelización

Incluye la creación de modelos del sistema que hay que desarrollar: estos modelos facilitarán la comprensión de los requisitos y el diseño del sistema. En cierto modo, esta técnica es equivalente a la construcción de maquetas o modelos en otras disciplinas de la ingeniería, con la diferencia de que, al ser el software un producto intangible, estos modelos no suelen tener una naturaleza física, sino que son intangibles como el propio software.

Ved también

Hablaremos en detalle de la modelización en el módulo "Análisis UML".

Actualmente, el lenguaje más utilizado para la creación de modelos de software es el lenguaje UML[1], que es un lenguaje publicado por el Object Management Group (OMG) y que define toda una serie de diagramas que nos permiten elaborar nuestros modelos.

(1)UML son las siglas de *unified modeling language*.

Uno de los motivos del éxito del lenguaje UML es que no impone ningún método de desarrollo y, por lo tanto, ha sido adoptado por la mayoría de los métodos de desarrollo actuales. Otra ventaja importante es que sólo define la notación que se debe usar pero no qué artefactos se han de generar, de manera que un mismo tipo de diagrama se puede usar en varios artefactos (modelo del dominio, diagrama de clases del diseño, etc.).

2.3.4. Construcción y pruebas

La construcción incluye la escritura del código fuente (implementación) y la realización de las pruebas necesarias para garantizar, dentro de lo posible, la ausencia de errores en los programas y la adecuación del producto a los requisitos.

Ved también

Las actividades de construcción y prueba y las siguientes no las trataremos en esta asignatura, las veremos en asignaturas posteriores, dentro del itinerario de ingeniería del software.

Una vez creado el código, habrá que crear los archivos ejecutables y ponerlos a disposición de los usuarios finales (lo que se denomina *despliegue*).

Como parte de esta actividad, también hay que tener en cuenta las tareas relacionadas con la gestión de la configuración (qué archivos forman parte de cada versión del sistema) y la gestión de los cambios (cómo aplicamos los cambios al código fuente para poder generar nuevas versiones del sistema de manera controlada).

2.3.5. Calidad

La gestión de la calidad es una actividad que incluye todas las etapas del desarrollo. Como mínimo, habrá que definir los criterios de aceptación del sistema: en qué momento decidiremos que el proyecto está finalizado satisfactoriamente.

Normalmente, la gestión de la calidad implica la recogida de métricas que nos ayuden a determinar si el software cumple o no los criterios de calidad marcados y también la documentación formal de los procesos de desarrollo, así como la verificación de su cumplimiento.

2.3.6. Mantenimiento y reingeniería

Tal y como hemos dicho anteriormente, una vez concluido el proyecto de desarrollo, habrá que poner en marcha nuevas actividades de operación y mantenimiento que quedarán fuera del ámbito de éste. Mientras que la operación no implica cambios en el software, el mantenimiento sí que lo hace.

El mantenimiento correctivo consiste en corregir los errores detectados en el software y está muy relacionado con la gestión de la configuración y del cambio, dado que, una vez detectado un error, habrá que corregirlo sobre el código fuente y generar una nueva versión del programa, que se deberá poner a disposición de los usuarios finales.

El mantenimiento evolutivo, en cambio, es más parecido a un proyecto de desarrollo, ya que consiste en añadir nuevas funcionalidades o adaptar las ya existentes para acomodar nuevas necesidades detectadas después del fin del desarrollo inicial.

2.3.7. Actividades desde el punto de vista del ciclo de vida del proyecto

El Project Management Institute elabora una clasificación diferente, ya que se fija en la gestión del proyecto y en qué punto del ciclo de vida se lleva a cabo cada tarea:

Project Management Institute

El Project Management Institute (PMI) es una asociación profesional de gestores de proyectos que se dedica, entre otras actividades, a publicar estándares de buenas prácticas en la gestión de proyectos.

- **Tareas de iniciación**. Son aquellas relativas a la toma de la decisión de si se empezará o no el proyecto (o la nueva fase del proyecto). Por ejemplo, una organización se puede plantear, como alternativa a un desarrollo, la adquisición de un producto ya desarrollado. Como resultado de las actividades de iniciación, obtendremos la definición del alcance del proyecto (lo que se hará y lo que no se hará como parte del proyecto) y la valoración de la duración (límite temporal) del proyecto y del coste (recursos humanos y materiales que habrá que invertir para llevar a cabo el proyecto).

- **Tareas de planificación**. Ayudan a organizar el trabajo, definiendo qué tareas habrá que llevar a cabo y en qué orden. Por ejemplo, se puede decidir planificar el proyecto en función de los riesgos (llevar a cabo primero las tareas que pueden poner en peligro la viabilidad del proyecto), por funcionalidades (desarrollar una funcionalidad de manera completa antes de trabajar en las otras) o por tipos de actividades (primero hacer todas las tareas de un tipo, después pasar a las del tipo siguiente, etc.).

- **Tareas de ejecución**. Son aquellas destinadas a completar el trabajo requerido para desarrollar el producto. Por ejemplo, escribir los programas.

- **Tareas de seguimiento y control**. Son las destinadas a observar la ejecución del proyecto para detectar los posibles problemas y adoptar las acciones correctivas que sean necesarias. También nos ayudan a validar la planificación (por ejemplo, validando que la estimación inicial de coste y tiempo era correcta).

- **Tareas de cierre**. Una vez finalizado el proyecto, hay que cerrarlo adecuadamente. Por ejemplo, habrá que realizar la aceptación del producto (que consiste en determinar que, efectivamente, el producto satisface los objetivos establecidos), resolver el contrato si se ha contratado una organización externa para hacer el desarrollo o traspasar el software al equipo de operaciones.

A pesar de que existe una evidente relación temporal entre los diferentes tipos de tareas, éstas no tienen lugar necesariamente de manera secuencial. De hecho, los diferentes métodos de desarrollo aconsejarán maneras diferentes de combinarlas, según cuál sea su ciclo de vida.

Mientras que un método puede aconsejar, por ejemplo, no empezar ninguna tarea de ejecución hasta que no se ha completado en gran medida la planificación, otros nos pueden aconsejar que no dediquemos demasiado esfuerzo a la planificación hasta que no hayamos ejecutado una parte del proyecto y hayamos alcanzado un cierto nivel de seguimiento y control.

2.4. Roles en la ingeniería del software

Tal y como hemos señalado anteriormente, todo método de desarrollo debe definir una serie de roles y también cuáles son sus responsabilidades (qué tareas ha de llevar a cabo cada rol). Esta definición de roles es muy importante, dado que el desarrollo de software es una actividad en la que interviene una gran variedad de personas y la interacción entre estas personas es la que determinará, finalmente, el éxito o fracaso del proyecto.

Los roles que defina un método de desarrollo dependerán, en parte, del alcance del método (si sólo se centra en la creación del software o si incluye también la parte más organizativa) y, en parte, también de los principios y valores que se quieran transmitir. Así, la organización en roles de los métodos que siguen el ciclo de vida en cascada es muy diferente de la de los métodos ágiles.

Lectura recomendada

Tom DeMarco; Timothy Lister (1999). *Peopleware: productive Projects and Teams* (2.ª ed.). Dorset House Publishing Company.

A continuación, estudiaremos algunos de los roles típicos que podemos encontrar (quizá con un nombre diferente) en cualquier método de desarrollo.

2.4.1. El jefe de proyectos

El PMBOK define la gestión de proyectos como "la aplicación del conocimiento, las habilidades, las herramientas y las técnicas a las actividades del proyecto para cumplir los requisitos del proyecto". La gestión del proyecto incluye, pues, tareas de iniciación, planificación, ejecución, seguimiento, control y cierre.

PMBOK

El PMBOK es el estándar publicado por el PMI que recoge el cuerpo de conocimiento ampliamente aceptado por sus miembros, es decir, recoge aquellas ideas con las que la mayoría de sus miembros está de acuerdo.

> El jefe de proyectos es la persona responsable de conseguir los objetivos del proyecto.

Ved también

Podéis encontrar más información sobre el PMBOK en el subapartado 5.4.

Por lo tanto, el jefe de proyectos es un rol no técnico encargado de organizar el proyecto, coordinar la relación entre el equipo de desarrollo y el resto de la organización y velar por el cumplimiento de sus objetivos tanto en relación con los costes, como con el valor generado.

2.4.2. Los expertos del dominio

Los expertos del dominio son las personas que conocen el dominio del sistema que se está desarrollando y, por lo tanto, son los principales conocedores de los requisitos. El caso más habitual es que no sean técnicos, sino que aporten su conocimiento sobre otras áreas, como ventas, finanzas, producción, etc.

Una vez más, los diferentes métodos de desarrollo demandarán diferentes niveles de implicación para los expertos del dominio. Mientras que siguiendo el ciclo de vida clásico éstos sólo son necesarios durante las fases iniciales, los métodos ágiles recomiendan que los expertos del dominio formen parte del equipo de desarrollo e incluso que se encuentren físicamente en la misma sala que los desarrolladores.

Sin embargo, a veces no podremos tener expertos del dominio para el desarrollo y se deberá recurrir a otras técnicas para la obtención de los requisitos, como las dinámicas de grupo, las entrevistas, etc.

Este rol también es conocido como analista de negocio[2], especialmente en entornos empresariales.

(2)En inglés, *business analyst*.

2.4.3. El analista funcional

El analista funcional es el responsable de unificar las diferentes visiones del dominio en un único modelo que sea claro, conciso y consistente. A pesar de que su tarea no es tecnológica, sí que es un perfil técnico, en el sentido de que debe conocer las notaciones y los estándares de la ingeniería del software para generar un modelo del sistema que sea adecuado para el uso por parte del resto de los técnicos.

Una diferencia importante respecto a los expertos del dominio es que conoce las limitaciones de la tecnología y las diferentes maneras de aplicarla de manera constructiva a la resolución de los problemas del negocio. Así, aunque quizá no conozca los detalles de la implementación del sistema, sí tiene una idea bastante clara de las posibilidades tecnológicas.

2.4.4. El arquitecto

El arquitecto es el responsable de definir las líneas maestras del diseño del sistema. Entre otras responsabilidades, tiene la de elegir la tecnología adecuada para la implementación del proyecto, para lo que debe tener en cuenta el tipo de producto, los conocimientos de los miembros del equipo y otros requisitos de la organización.

Por lo tanto, el arquitecto es un rol técnico con un buen conocimiento de las tecnologías de implementación. A partir de los requisitos recogidos por el analista, creará un conjunto de documentos de arquitectura y diseño que el resto de desarrolladores usará como base para su trabajo.

En algunos métodos (especialmente en los incrementales), el arquitecto también es responsable de implementar la arquitectura y de validar la viabilidad y la idoneidad.

2.4.5. El analista orgánico o analista técnico

El analista orgánico se encarga del diseño detallado del sistema respetando la arquitectura definida por el arquitecto. El destinatario de su trabajo es el programador.

El analista orgánico tomará como punto de partida un subconjunto de los requisitos del analista y la definición de la arquitectura y diseñará las partes del sistema que implementan los requisitos mencionados hasta un nivel de detalle suficiente para la implementación.

En muchos métodos de desarrollo, especialmente en los que tienden menos a la especialización, no existe un rol diferenciado para el analista orgánico, sino que los programadores, mencionados a continuación, desempeñan también este rol.

2.4.6. Los programadores

Los programadores son los responsables de escribir el código fuente a partir del cual se han de generar los programas. Por lo tanto, son expertos en la tecnología de implementación. Para llevar a cabo su tarea, partirán de los diseños detallados creados por el analista orgánico.

A pesar de que el término *programador* sólo se aplica a las personas que generan código fuente en un lenguaje de programación, también podríamos incluir en este grupo a otros especialistas, como los expertos en bases de datos que se encargarán de definir la estructura de las tablas y de escribir las consultas en la base de datos.

2.4.7. El experto de calidad (probador)

Su trabajo es velar por la calidad del producto desarrollado. A pesar de que los desarrolladores (entendiendo como desarrolladores a los arquitectos, analistas orgánicos y programadores) han de velar por la calidad del código fuente y deben crear pruebas automatizadas, el experto de calidad complementa esta tarea y aporta un punto de vista externo.

Por lo tanto, el experto de calidad debe partir de los requisitos y de los programas y ha de verificar que, efectivamente, los programas cumplen los requisitos establecidos. Su responsabilidad es muy importante: es quien debe decidir si el sistema se puede poner o no en manos de los usuarios finales.

2.4.8. El encargado del despliegue

Una vez creados y validados los programas, el encargado del despliegue los ha de empaquetar y enviarlos a los entornos adecuados para que lleguen a manos de los usuarios finales.

Su trabajo, pues, está muy relacionado con las actividades de gestión de la configuración y de los cambios.

2.4.9. El responsable de producto

Otro rol muy importante en algunos ámbitos es el del responsable de producto. El responsable de producto es la persona que tiene la visión global del producto que se quiere desarrollar y vela por su desarrollo correcto.

La tarea del responsable de producto no consiste en hacer un seguimiento detallado del proyecto (esto lo hace el jefe de proyecto), sino en aportar una visión global del producto y asegurarse de que el proyecto (o proyectos, si hay más de uno relacionado) encaja perfectamente con los objetivos y la estrategia de la organización.

El responsable de producto no ha de ser necesariamente un experto del dominio, pero en cambio sí debe conocer perfectamente la estrategia de la organización y los motivos por los que se está desarrollando el software.

2.4.10. Otros roles

La lista de roles que hemos mencionado no es exhaustiva, ya que, como se ha dicho antes, los roles dependerán, en gran medida, de cómo organice el desarrollo el método concreto que queramos aplicar.

También debemos tener en cuenta que habrá muchas personas que tendrán una cierta influencia sobre el proyecto, aunque no formen parte propiamente del equipo responsable del desarrollo, como podría ser el caso del jefe de desarrollo, el director general de la organización, etc.

3. Métodos de desarrollo de software

Hemos visto hasta ahora que, dentro del ámbito de la ingeniería del software, la actividad de desarrollo se suele organizar a modo de proyectos y que la manera de gestionar estos proyectos estará determinada por el método de desarrollo que queramos aplicar.

El método de desarrollo definirá un ciclo de vida (qué etapas forman parte del proyecto de desarrollo), qué procesos, actividades y tareas tienen lugar en las diferentes etapas del ciclo de vida, quién se encargará de llevar a cabo cada una de las tareas y también la interacción entre tareas, roles y personas.

En este apartado, veremos una clasificación de los métodos de desarrollo que nos ayudará a situarlos en contexto y a elegir un método u otro según las características del producto que hay que desarrollar. También estudiaremos algunas de las familias de métodos más difundidas actualmente: el ciclo de vida clásico o en cascada, el desarrollo iterativo e incremental y el desarrollo ágil/*lean*.

3.1. Historia de los métodos de desarrollo

La gestión de proyectos es una disciplina que se ha estudiado ampliamente a lo largo de los años (especialmente a partir de los años cincuenta), lo que ha dado lugar a toda una serie de modelos que nos indican cómo debemos llevar a cabo la gestión de los proyectos. Probablemente, el modelo de gestión de proyectos más difundido hoy en día es el del Project Management Institute (PMI).

Una de las primeras metáforas que se aplicaron en el desarrollo de software es la gestión científica (*scientific management*, desarrollada por Frederik Taylor a finales del siglo XIX). A grandes rasgos, la gestión científica consiste en descomponer el proceso industrial en un conjunto de pequeñas tareas lo más repetitivas posible que puedan ser ejecutadas por un trabajador altamente especializado, cuyo rendimiento debe ser fácilmente medible por los encargados de gestionar el proceso.

El ejemplo clásico de gestión científica es la cadena de producción, que permitió reducir espectacularmente el coste de la producción masiva en serie de infinidad de productos industriales. Por lo tanto, es natural que fuera uno de los primeros métodos industriales que se intentaron aplicar al desarrollo de software y que dio lugar al ciclo de vida en cascada (que veremos más adelante).

No obstante, con el paso del tiempo se fue abandonando este enfoque en favor de otros más flexibles, tanto en la industria del desarrollo de software como en la de manufactura. Uno de los métodos de gestión de mayor éxito ha sido el método de producción Toyota, que ha dado lugar a lo que se denomina *lean-manufacturing*.

Enlace de interés

Toyota Production System http://www2.toyota.co.jp/en/vision/production_system/index.html.

El método de producción Toyota se basa en dos principios:

- *Jidoka*: evitar producir productos defectuosos parando, si es necesario, la línea de producción y
- la producción *just-in-time*: producir sólo aquellos productos que son necesarios en la fase siguiente y no acumular excedentes.

La aplicación de esta filosofía al desarrollo de software es lo que se conoce como *lean software development*.

Hay estudios (Cusumano, 2003) que han demostrado que, si bien un trabajo previo de planificación y descomposición del problema es una buena manera de conseguir un sistema con pocos defectos, otros métodos que ponen énfasis en la obtención de información basada en observaciones hechas sobre un sistema real y en la adaptación a los cambios pueden conseguir una tasa de defectos similar y además permiten adaptarse mejor a los cambios requeridos por los usuarios.

La diversidad de métodos de desarrollo es tan grande que, a finales del 2009, un grupo de especialistas en ingeniería del software formado, entre otros, por gente tan distinta como Erich Gama, Ivar Jacobson, Ken Schwaber o Edward Yourdon, formaron el grupo SEMAT[3] con la finalidad (entre otras) de unificar el campo de la ingeniería del software.

[(3)]SEMAT son las siglas de *software engineering method and theory*.

El manifiesto del SEMAT dice: "La ingeniería del software está gravemente dificultada hoy en día por prácticas inmaduras. Los problemas específicos incluyen:

- La prevalencia de modas pasajeras más típicas de la industria de la moda que de una disciplina de la ingeniería.
- La carencia de una base teórica sólida y ampliamente aceptada.
- El enorme número de métodos y variantes de métodos, con diferencias pobremente comprendidas y magnificadas artificialmente.
- La carencia de evaluación y validación experimental creíble.
- La separación entre la práctica en la industria y la búsqueda científica".

3.2. Clasificación de los métodos de desarrollo

A pesar de que las actividades por llevar a cabo son, a grandes rasgos, las mismas independientemente del método, las diferencias respecto a cómo y cuándo se deben realizar darán lugar a métodos absolutamente diferentes, con características muy diferenciadas.

Así, por ejemplo, hay métodos que aconsejarán elaborar un análisis exhaustivo y formal de los requisitos antes de empezar ninguna actividad de construcción, mientras que otros aconsejarán pasar a la construcción una vez que se tengan los requisitos sin llevar a cabo ningún tipo de modelo.

Por lo tanto, una de las primeras tareas del ingeniero de software será elegir el método de desarrollo más adecuado a la naturaleza del proyecto que se deba llevar a cabo; algunos de los factores que se suelen usar para clasificar los proyectos son (Wysocki, 2009):

- riesgo
- valor de negocio
- duración (menos de tres meses, de tres a seis meses, más de seis meses, etc.)
- complejidad
- tecnología utilizada
- número de departamentos afectados
- coste

Así, no seguiremos el mismo proceso para desarrollar un proyecto corto (unos tres meses) con poco riesgo (es decir, las circunstancias en las que se ha de desarrollar el proyecto son previsibles), que aporte un valor relativamente pequeño y poco complejo, que para desarrollar un proyecto de tres años, en el que hay mucha incertidumbre y que debe ser la herramienta fundamental de trabajo para nuestra organización.

Simplificando esta clasificación, podemos situar cada uno de nuestros proyectos en uno de estos cuatro grupos (Wysocki, 2009) según si tenemos clara la necesidad que queremos cubrir (objetivo) y si conocemos o no los detalles de cómo será la solución (requisitos, tecnología, etc.).

	Solución conocida	**Solución poco conocida**
Objetivo claro	1	2
Objetivo poco claro	3	4

En el grupo 1 encontramos los proyectos para los que está claro qué queremos hacer y cómo lo haremos. En este caso, podremos elegir un método con poca tolerancia al cambio, pero que, en cambio, sea sencillo de aplicar.

Por el contrario, para los proyectos del grupo 2 necesitaremos un método que nos permita cambiar las ideas iniciales a medida que el proyecto avance y que facilite el descubrimiento de la solución mediante ciclos de retroalimentación. Actualmente, la mayoría de los proyectos pertenecen a este grupo.

En el grupo 3 encontramos un tipo de proyecto bastante peculiar, dado que se trata de proyectos en los que tenemos la solución pero todavía debemos buscar el problema.

Ejemplo del grupo 3

Un ejemplo del grupo 3 sería un proyecto en el que queremos evaluar un producto existente para ver si cubre alguna de las necesidades de la organización. Así, por ejemplo, queremos promover la introducción del software libre en nuestra organización pero no sabemos en qué áreas sacaremos el máximo provecho.

Finalmente, en el grupo 4 encontraríamos, por ejemplo, los proyectos de investigación y desarrollo, en los que debemos ser flexibles tanto respecto a la solución final que encontramos como respecto al problema que solucionamos, dado que, muchas veces, se acaba solucionando un problema diferente.

Post It

Un ejemplo de proyecto de tipo 4 no relacionado con el software es el de las notas adhesivas Post It. En 1968, los científicos de 3M desarrollaron un nuevo tipo de pegatina poco potente pero reutilizable, aunque no tuvieron mucho éxito en promocionarlo hasta que, en 1974, otro equipo decidió usarlo para pegar notas adhesivas y creó un producto totalmente diferente del que el equipo inicial tenía en mente.

Los proyectos de desarrollo suelen tener un objetivo bastante claro y, por lo tanto, pertenecer a los grupos 1 o 2. Como se puede ver, la manera de gestionar unos y otros proyectos variará enormemente y, por ello, es importante que los ingenieros de software conozcan diferentes métodos de desarrollo y puedan elegir, en cada momento, el más adecuado.

Nosotros clasificaremos los métodos de desarrollo en tres grandes familias:

- los que siguen el ciclo de vida en cascada (adecuado para los proyectos de tipo 1) por un lado,
- los métodos iterativos e incrementales y
- los ágiles, por el otro lado (más adecuados para los de tipo 2).

Métodos ágiles

A pesar de que la mayoría de los métodos ágiles siguen el ciclo de vida iterativo e incremental, los hemos considerado una familia diferente debido a sus peculiaridades.

3.2.1. Ciclo de vida clásico o en cascada

El ciclo de vida clásico o en cascada es ideal para proyectos del grupo 1, dado que es muy sencillo de aplicar, pero en cambio es poco tolerante a los cambios. La manera de organizar el desarrollo es muy similar a una cadena de producción: tenemos trabajadores altamente especializados (analista funcional, analista orgánico, programador, especialista en pruebas, arquitecto, etc.) que

producen artefactos que son consumidos por otros trabajadores de la cadena como parte del proceso global de desarrollo (el analista funcional produce requisitos que el analista orgánico transforma en diseños que el programador transforma en código, etc.).

El producto pasa, progresivamente, por las etapas siguientes:

- **Requisitos**. Definir qué debe ser el producto que hay que desarrollar. Dado que el ciclo de vida en cascada es poco flexible a los cambios, esta etapa es crítica para el éxito del proyecto, pues un defecto en los requisitos se propagaría por el resto de las etapas y amplificaría los efectos nocivos. Para evitar este problema, los diferentes métodos dispondrán de herramientas como las revisiones de requisitos o los lenguajes formales de especificación como el OCL[4].

- **Análisis y diseño**. Definir cómo debe ser el producto tanto desde el punto de vista externo como interno. El análisis da un punto de vista externo documentando mediante modelos sobre qué hace el sistema, mientras que el diseño da el punto de vista interno documentando cómo obra (qué componentes forman parte de él, cómo se relacionan entre ellos, etc.).

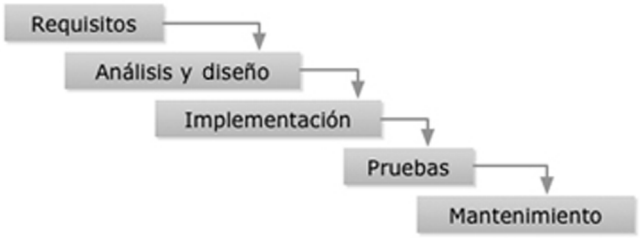
- **Implementación**. Escribir el código, los manuales y generar el producto ejecutable. Este código se debe escribir según las indicaciones efectuadas en la fase de análisis y diseño.

- **Pruebas**. Se verifica que el producto desarrollado se corresponda con los requisitos. En este punto se muestra el producto a los usuarios finales para que validen el resultado.

- **Mantenimiento**. Se pone el sistema a disposición de todos los usuarios y se corrigen los defectos que se vayan encontrando.

[4]OCL son las siglas de *object constraint language*.

Ciclo de vida en cascada



Para mejorar la tolerancia a los cambios y la adaptabilidad de estos métodos, se pueden establecer ciclos de retroalimentación en el análisis final de cada etapa. Así, por ejemplo, al finalizar la etapa de análisis y diseño, podemos revisar los requisitos para incorporar las correcciones a los errores que hayamos ido encontrando durante el análisis y el diseño.

En cualquier caso, el ciclo de vida en cascada se caracteriza por su carácter secuencial. Esta naturaleza favorece, tal y como se ha dicho anteriormente, la especialización de los miembros del equipo de desarrollo en una sola actividad concreta, lo que facilita, a su vez, la posibilidad de tener equipos diferentes especializados en tareas diferentes. Así, por ejemplo, se favorece tener analistas que se han especializado en modelización y programadores que se han especializado en construcción.

3.2.2. Ciclo de vida iterativo e incremental

El ciclo de vida en cascada tiene el inconveniente (especialmente para proyectos grandes) de que no tenemos ningún tipo de información empírica sobre el producto final hasta que no estamos a punto de finalizar el proyecto. Aunque se hayan hecho revisiones de todos los artefactos, el conocimiento íntegro que tenemos sobre la marcha del proyecto y sobre el producto final es teórico.

Cuando llega el momento en el que tenemos una versión del sistema que podemos probar para obtener datos empíricos, ya es muy tarde para solucionar los posibles errores introducidos en las primeras fases del desarrollo. Por lo tanto, si seguimos este ciclo de vida, el coste de los errores en las primeras etapas es mucho mayor que el coste de los errores en las etapas finales. Por ello, se considera poco adecuado para los proyectos del grupo 2, en los que no sabemos, a priori, cómo será el resultado final del desarrollo y, por lo tanto, es muy fácil que cometamos errores durante las fases iniciales.

Para este tipo de proyectos, necesitaremos un método que nos permita cambiar las ideas iniciales a medida que el proyecto avance y que facilite el descubrimiento de la solución mediante la obtención de información empírica lo mejor posible: es el caso de los métodos iterativos e incrementales, como UP o los métodos ágiles.

Ved también

Veremos el proceso unificado (UP) en el subapartado 3.3.2 de este módulo didáctico.

Otro problema del ciclo de vida en cascada que intentan solucionar los métodos iterativos es la obtención de resultados parciales utilizables. Puesto que debemos esperarnos a tener todos los requisitos, el análisis y el diseño antes de empezar la implementación, podemos estar mucho tiempo (especialmente en proyectos largos) invirtiendo en el desarrollo sin recuperar, ni siquiera parcialmente, la inversión realizada.

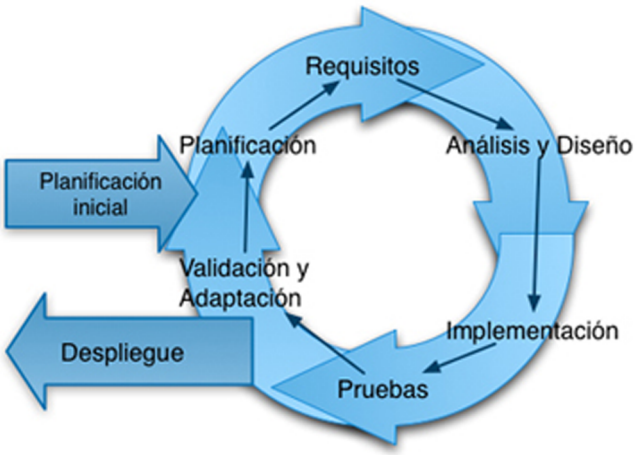
Un método iterativo organiza el desarrollo en una serie de iteraciones y cada una de ellas es un miniproyecto autocontenido que amplía el resultado final de la iteración anterior. Por otro lado, un método incremental nos asegura que al final de la iteración tenemos un resultado final utilizable. Así, el ciclo de vida iterativo e incremental evita los dos problemas mencionados anteriormente:

- **Acelera la retroalimentación**. Dado que cada iteración cubre todas las actividades del desarrollo (requisitos, análisis, implementación y pruebas), tenemos información sobre todos los ámbitos del producto desde el prin-

cipio y, por ejemplo, si la arquitectura que hemos definido no se puede implementar, lo veremos al principio y no al final del proyecto. También nos permite obtener información empírica basada en el producto real, dado que la parte que está desarrollada es ya definitiva.

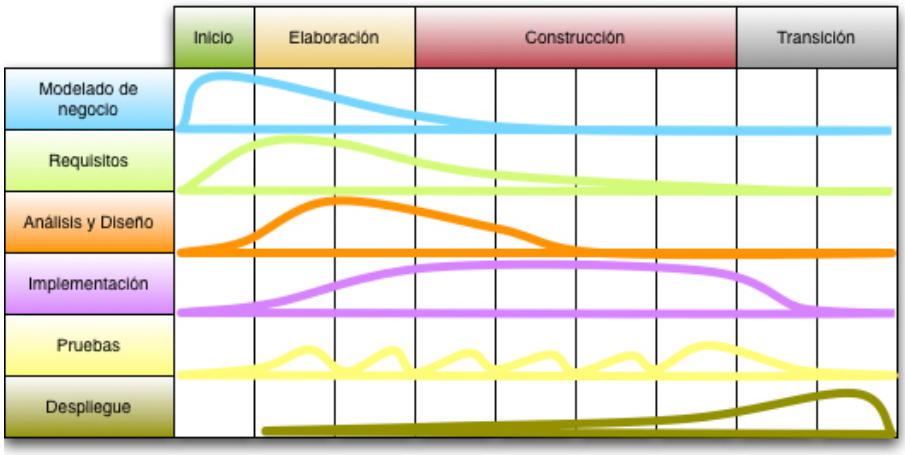
- **En todo momento se tiene un producto operativo**. Como se van añadiendo funcionalidades a un producto operativo, en cualquier momento se puede decidir usar el sistema desarrollado, lo que permite acelerar el regreso de la inversión o, incluso, finalizar el proyecto antes de consumir todos los recursos asignados si se llega a un punto en el que el producto es "suficiente bueno".

Ciclo de vida iterativo



Normalmente, una iteración durará entre una y seis semanas. Para facilitar la extrapolación de resultados (por ejemplo, en volumen de trabajo efectuado en cada iteración), todas las iteraciones suelen tener la misma longitud (es lo que se denomina *time-boxing*). En este tiempo, hay que trabajar en todas las actividades del desarrollo (planificación, requisitos, análisis y diseño, implementación, pruebas, validación) a pesar de que, a lo largo del tiempo, el énfasis que damos a cada una de las actividades irá variando (las primeras iteraciones pondrán más énfasis en los requisitos y el análisis, mientras que las últimas lo harán en las pruebas). Al final de cada iteración, habrá que adaptar el proceso y la planificación en función del conocimiento generado durante la iteración.

Énfasis sobre las diferentes actividades en el proceso unificado (UP)



Ved también

Estudiaréis con más detalle el proceso unificado en el subapartado 3.3.2.

No obstante, hay que tener en cuenta que cada iteración debe ser un proyecto completo y, por lo tanto, ha de incluir todas las etapas del ciclo de vida en cascada (menos el mantenimiento). Uno de los errores habituales en la aplicación del ciclo de vida iterativo e incremental consiste en disfrazar de iterativo el ciclo de vida en cascada y considerar cada una de sus etapas como una iteración (haciendo, pues, una iteración –o varias– de requisitos, y después iteraciones de análisis, iteraciones de diseño, etc.).

El desarrollo iterativo es incremental porque al final de cada iteración se produce un incremento en el volumen de funcionalidad implementada en el sistema. Por lo tanto, el producto final no se construye de golpe al final del proyecto, sino que se va construyendo a medida que avanzan las iteraciones.

El desarrollo iterativo e incremental también facilita la planificación centrada en el cliente y en los riesgos y avanza la implementación de aquellas funcionalidades que implican más riesgos (por ejemplo, integrar una tecnología desconocida) y aquellas que ofrecen más valor para el cliente. Como parte del proceso de adaptación, los *stakeholders* suelen tener la posibilidad de elegir qué funcionalidades se implementarán en cada iteración.

3.2.3. Desarrollo *lean* y ágil

El desarrollo ágil[5] es un conjunto de métodos de desarrollo iterativos que comparten unos principios que se recogieron en el Manifiesto ágil del 2001.

(5)En inglés, *agile software development*.

Manifiesto por el desarrollo ágil del software

Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

- **Individuos e interacciones** sobre procesos y herramientas.
- **Software funcionando** sobre documentación extensiva.
- **Colaboración con el cliente** sobre negociación contractual.
- **Respuesta ante el cambio** sobre seguir un plan.

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.

Fuente: http://www.agilemanifesto.org/iso/es/

Hay que tener en cuenta que, en la frase final, se reconoce el valor de los elementos de la derecha: los métodos ágiles no están en contra de estos elementos, sino que los consideran secundarios en relación con los otros.

Ejemplo

Por ejemplo, los métodos ágiles consideran que las personas que participan en un proyecto (y la manera como interactúen) son más decisivas para el éxito que los procesos que se aplican o las herramientas que usan. Por ello, suelen ser métodos poco estrictos en cuanto a los procesos que se deben seguir y los artefactos que se han de generar, por lo que se denominan también *métodos ligeros*.

Desde el punto de vista de la ingeniería del software, este principio parecería ir en contra del enfoque sistemático, dado que asume que un proceso, por muy definido que esté, siempre dependerá de las personas y, por lo tanto, limita la transferibilidad del conocimiento entre diferentes ejecuciones del mismo proceso; sin embargo, en la práctica ocurre que el proceso definido en los métodos ágiles se centra más en las personas y en sus interacciones que en los roles y en las tareas que éstas asumen. No se trata, pues, de no tener ningún proceso definido, sino de que el proceso se centre en la interacción entre personas.

Estos principios encajan perfectamente con la filosofía del desarrollo iterativo e incremental, ya que los métodos iterativos e incrementales dan prioridad al software operativo (el producto final de cada iteración es una versión operativa del software) y facilitan la colaboración con el cliente y la respuesta al cambio, además permiten cambios en la planificación de las diferentes iteraciones (se puede decidir priorizar una funcionalidad o cambiarla por otra sin

afectar significativamente a la planificación general del proyecto). De hecho, a pesar de que el manifiesto ágil no hace ninguna referencia al ciclo de vida, la mayoría de los métodos ágiles son iterativos e incrementales.

Otra diferencia importante respecto al ciclo de vida en cascada es que, dado que con los métodos ágiles el coste de un cambio en la implementación es muy menor al del ciclo de vida en cascada, se da menos importancia al trabajo previo de análisis y se potencia el cambio y la adaptación.

El desarrollo *lean*, como se ha dicho anteriormente, hace referencia a las técnicas de manufactura *lean* derivadas del sistema de producción de Toyota (el *Toyota Production System*) y se popularizó en el ámbito de la ingeniería de software a raíz de la publicación del libro *Lean Software Development: An Agile Toolkit*, de Mary Poppendieck y Tom Poppendieck (2003). En este libro, los autores explican la filosofía de manufactura *lean* y, al mismo tiempo, extrapolan los principios y las prácticas *lean* al desarrollo de software, relacionándolo con el desarrollo ágil. La filosofía *lean* se basa en siete principios:

1) Evitar la producción innecesaria. En la línea del concepto de producción *just-in-time*, se trata de producir sólo aquello que se necesita en la etapa siguiente. Así, en vez de producir todos los requisitos antes de elaborar el análisis, se producen sólo los necesarios para poder hacer el análisis de esta iteración. En esta misma línea, también debemos concentrarnos en producir sólo los artefactos que realmente aportan valor; a grandes rasgos, lo podríamos resumir con la pregunta: "¿hay alguien esperando este artefacto?" o "¿alguien se quejará si este artefacto no está actualizado?". Sorprendentemente, muchas veces la respuesta a esta pregunta es no.

2) Amplificar el aprendizaje. Hay que recopilar toda la información que se va generando sobre el producto y su producción (retroalimentación, iteraciones cortas, etc.) para entrar en un ciclo de mejora continua de la producción.

3) Decidir lo más tarde posible. Intentar tomar todas las decisiones con el máximo de información posible. Esto significa atrasar cualquier decisión hasta el punto de que no tomar la decisión resulta más caro que tomarla.

> Por ejemplo, si un estudiante puede decidir anular la convocatoria de un examen hasta tres días antes del examen, pero no sabe si tendrá tiempo de estudiar, debería tomar la decisión tres días antes del examen, dado que si la toma antes no tendrá tanta información sobre si lo puede aprobar o no como en aquel momento; pero si no la toma en aquel momento, ya será demasiado tarde para decidir.

4) Entregar el producto cuanto antes mejor. Es necesario que el proceso de desarrollo nos permita implementar rápidamente los cambios y las funcionalidades que queramos añadir al sistema de información.

5) Dar poder al equipo. Es importante que los equipos de personas que forman parte del desarrollo se sientan responsables del producto y, por lo tanto, hay que evitar que haya un gestor que tome todas las decisiones. El equipo de desarrollo es quien decide sobre las cuestiones técnicas y quien se responsabilizan de cumplir los plazos.

6) Incorporar la integridad. El software se debe mantener útil a lo largo del tiempo, adaptándose si es necesario a nuevos requisitos, por lo que hay que desarrollarlo de manera que se pueda cambiar fácilmente.

7) Visión global. Hay que evitar las optimizaciones parciales del proceso que pueden causar problemas en otras etapas y adoptar una visión global del sistema. Por ejemplo, si decidimos eliminar una funcionalidad por motivos técnicos, hay que tener en cuenta cómo puede afectar esto al valor del producto desde el punto de vista de la organización. Este principio va en contra de los trabajadores hiperespecializados que proponían el ciclo de vida en cascada.

Los principios *lean* encajan perfectamente con los principios del Manifiesto ágil (de hecho, los podríamos considerar un superconjunto de los principios ágiles), a pesar de que, a diferencia del Manifiesto ágil, muy centrado en el punto de vista de los desarrolladores, la filosofía *lean* intenta integrar el desarrollo de software dentro de un marco conceptual y un conjunto de prácticas aplicadas a otras industrias y que afectan a todos los aspectos de la organización (no sólo al desarrollo).

3.3. Ejemplos de métodos de desarrollo

Una de las conclusiones de Cusumano (2003) es que las prácticas para organizar el desarrollo no se podían adoptar de manera aislada, sino que deben formar parte de un todo coherente que nos ayude a neutralizar los efectos negativos de algunas prácticas. Por ejemplo, si no hacemos un modelo de requisitos muy detallado antes de empezar el desarrollo, es importante que mostremos un prototipo o una versión beta del sistema a los usuarios finales para poder asegurarnos de que el sistema desarrollado satisface correctamente sus necesidades y de que podremos hacer cambios con poco coste si el producto no satisface los requisitos reales de los usuarios.

Para conseguir esta coherencia entre prácticas, se desarrollaron los métodos de desarrollo, que –tal y como hemos dicho anteriormente– no sólo incluyen el ciclo de vida, sino que también dan indicaciones más detalladas sobre los roles, las actividades, los artefactos y las prácticas que se deben aplicar en el desarrollo de software.

A continuación describimos tres:

1) uno que sigue el ciclo de vida en cascada (Métrica 3),

2) uno iterativo e incremental (el proceso unificado o UP),

3) uno ágil (Scrum).

La finalidad de este subapartado no es dar una descripción pormenorizada de los métodos, sino ofrecer una visión general que permita realizar un análisis comparativo entre éstos.

3.3.1. Métrica 3

El método Métrica 3 lo desarrolló el Ministerio de Administraciones Públicas del Estado español con el fin de servir de referencia a todas las administraciones públicas del Estado.

Nota

Métrica 3 se refiere a la versión 3 de Métrica, que se publicó el año 2000. La primera versión se publicó en 1989.

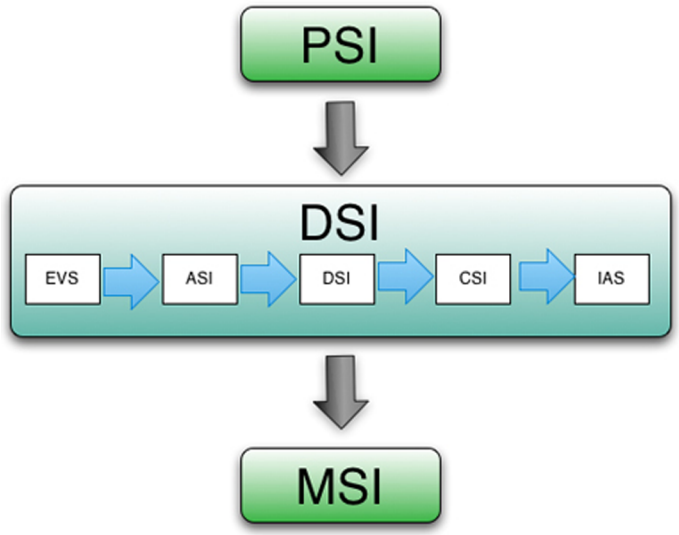
Métrica 3 distingue tres procesos; el segundo incluye cinco subprocesos:

1) planificación de sistemas de información (PSI)

2) desarrollo de sistemas de información (DSI)
- **a)** estudio de viabilidad del sistema (EVS)
- **b)** análisis del sistema de información (ASI)
- **c)** diseño del sistema de información (DSI)
- **d)** construcción del sistema de información (CSI)
- **e)** implantación y aceptación del sistema (IAS)

3) mantenimiento de sistemas de información (MSI)

Los procesos de Métrica 3



Como podemos ver, Métrica 3 incluye procesos más allá del de desarrollo propiamente dicho y regula tanto el proceso previo de planificación como el proceso posterior de mantenimiento.

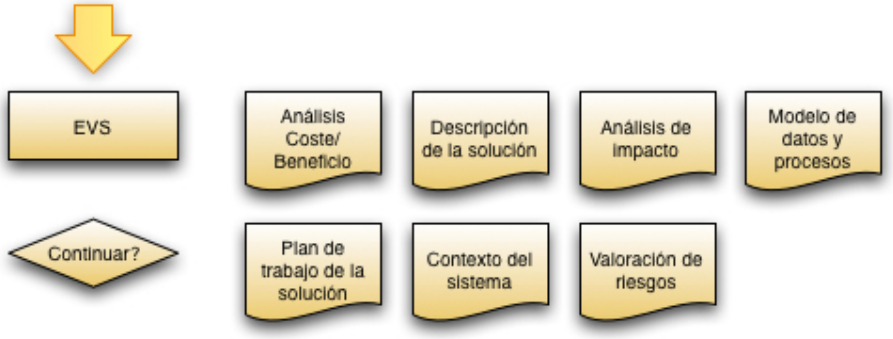
El plan de sistemas de información tiene como finalidad asegurar que el desarrollo de los sistemas de información se realiza de manera coherente con la estrategia corporativa de la organización. El plan de sistemas de información, por lo tanto, es la guía principal que deben seguir todos los proyectos de desarrollo que empiece la organización. Como tal, incorpora un catálogo de requisitos que deben cumplir todos los sistemas de información, una arquitectura tecnológica, un modelo de sistemas de información, etc.

Métrica 3: artefactos y tareas del proceso PSI

El proceso de desarrollo del sistema de información propiamente dicho empieza con el estudio de viabilidad del sistema (EVS). Este estudio responde a la pregunta de si es necesario o no continuar con el desarrollo del sistema. Para dar esta respuesta, tendrá en cuenta criterios económicos, legales, técnicos y operativos.

Como resultado del estudio de viabilidad del sistema obtendremos, por ejemplo, un análisis coste/beneficio de la solución, una planificación, una descripción de la solución, un modelo de descomposición en subsistemas, etc. Según el tipo de proyecto (desarrollo o implantación de un producto estándar) tendremos otros artefactos, como el modelo de datos, el modelo de procesos o la descripción del producto, un análisis de costes del producto, etc.

Métrica 3: estudio de viabilidad del sistema



Etapas del ciclo de vida en cascada

A continuación, se llevarán a cabo las etapas típicas del ciclo de vida en cascada: análisis del sistema de información (ASI), diseño del sistema de información (DSI), construcción del sistema de información (CSI) e implantación y aceptación del sistema de información (IAS).

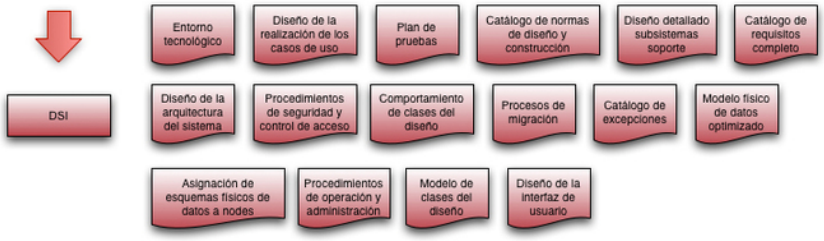
El análisis del sistema de información (ASI) tiene la finalidad de conseguir la especificación detallada del sistema de información, mediante un catálogo de requisitos y una serie

de modelos que cubran las necesidades de información de los usuarios para los que se desarrollará el sistema de información. En este proceso, también se inicia la especificación del plan de pruebas que se completará en la fase siguiente (DSI).

Métrica 3: análisis del sistema de información

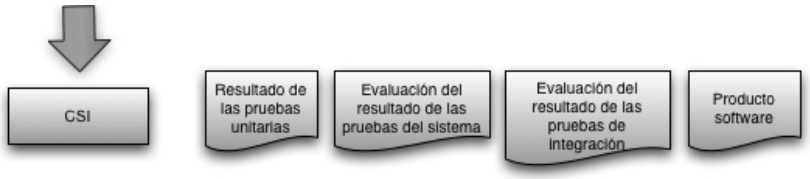
El propósito del diseño del sistema de información es obtener la definición de la arquitectura del sistema y del entorno tecnológico que le debe apoyar, junto con la especificación detallada de los componentes del sistema de información.

Métrica 3: diseño del sistema de información



La construcción del sistema de información (CSI) tiene como objetivo la construcción y prueba de los diferentes componentes del sistema de información, a partir del conjunto de especificaciones lógicas y físicas de éste, obtenido en el proceso de diseño del sistema de información (DSI). Se desarrollan los procedimientos de operación y seguridad y se elaboran los manuales de usuario final y de explotación, estos últimos cuando proceda.

Métrica 3: construcción del sistema de información

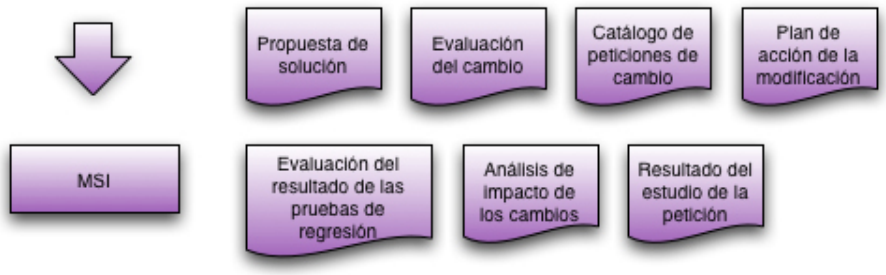


El proceso de implantación y aceptación del sistema (IAS) tiene como objetivo la entrega y la aceptación del sistema en su totalidad, que puede comprender varios sistemas de información desarrollados de manera independiente, según se haya establecido en el proceso de estudio de viabilidad del sistema (EVS); un segundo objetivo es llevar a cabo las actividades oportunas para el paso a producción del sistema.

Métrica 3: implantación y aceptación del sistema

A partir de este momento, se considera que el desarrollo del SI está finalizado y empieza la última etapa: el proceso de mantenimiento del sistema de información (MSI).

Métrica 3: mantenimiento del sistema de información



3.3.2. Proceso unificado

El proceso unificado (UP, *unified process*) fue propuesto por la empresa Rational Software (actualmente parte de IBM) con la intención de ser para los métodos de desarrollo de software lo que el UML[6] fue para los lenguajes de modelización: una versión unificada de los diferentes procesos existentes que incorporara las mejores prácticas existentes. Intenta mejorar, respecto al ciclo de vida en cascada por la vía de mejorar la productividad de los equipos, el uso extensivo de modelos y la adopción de un ciclo de vida iterativo e incremental.

(6)UML son las siglas de *unified modeling language*.

Nota

El primer libro sobre UP se publicó en 1999 con el nombre *The Unified Software Development Process*. Los autores (Ivar Jacobson, Grady Booch y James Rumbaugh) son considerados los padres del proceso unificado.

UP es un bastimento de procesos configurable que se puede adaptar a diferentes contextos, de manera que no se aplica igual al desarrollo de una intranet de una empresa que al desarrollo de software militar. En nuestro caso, nos centraremos en la variante OpenUP para la descripción de tareas y roles, dado que es una de las más sencillas y, además, está disponible públicamente. A diferencia de otras variantes como RUP[7], OpenUP asume los valores del desarrollo ágil y los tiene en cuenta a la hora de personalizar el proceso unificado.

(7)RUP son las siglas de *rational unified process*.

Sin embargo, en términos generales todas las variantes del proceso unificado se basan en potenciar seis prácticas fundamentales (según el *rational unified process: best practices for software development teams*):

1) Desarrollo iterativo e incremental

2) Gestión de los requisitos (mediante casos de uso)

3) Arquitecturas basadas en componentes (módulos o subsistemas con una función clara)

4) Utilización de modelos visuales (mediante el lenguaje UML)

5) Verificación de la calidad del software (a lo largo de todas las etapas del proceso, no sólo al final)

6) Control de los cambios al software

Además, UP promueve una gestión proactiva de los riesgos y sugiere que se construyan antes las partes que incluyan más riesgos, así como el fomento de la construcción de una arquitectura ejecutable durante las primeras etapas del proyecto.

El proceso se puede descomponer de dos maneras: en función del tiempo (en fases) y en función de las actividades o contenido (actividades). Además, tal y como hemos dicho que debe hacer cualquier método, definirá los roles que tienen las diferentes personas y qué tareas ha de llevar a cabo cada rol.

Fases del proyecto

A lo largo del tiempo, el proyecto pasará (de manera secuencial) por cuatro fases:

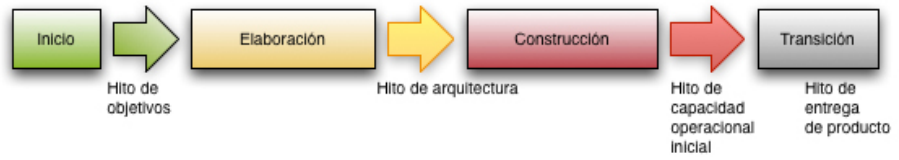
1) Inicio (*inception*). Se analiza el proyecto desde el punto de vista de la organización y se determina el ámbito, la valoración de los recursos necesarios, la identificación de riesgos y de casos de uso.

2) Elaboración (*elaboration*). Se analiza el dominio del sistema que hay que desarrollar hasta tener un conjunto de requisitos estable, se eliminan los riesgos principales y se construye la base de la arquitectura (ésta será ejecutable).

3) Construcción (*construction*). Se desarrolla el resto de la funcionalidad (de manera secuencial o en paralelo) y se obtiene, como resultado, el producto y la documentación.

4) Transición (*transition*). Se pone el producto a disposición de la comunidad de usuarios. Esto incluye las pruebas beta, la migración de entornos, etc.

Fases e hitos del proceso unificado



Al final de cada fase, se llega a un hito en el que se deben poder responder a una serie de preguntas y decidir si se continúa o no con el proyecto.

La serie de objetivos analiza el coste y los beneficios esperados del proyecto; debemos ser capaces de responder a dos preguntas: ¿estamos de acuerdo sobre el ámbito (qué debe hacer y qué no debe hacer el sistema) y los objetivos del proyecto? ¿Estamos de acuerdo sobre si vale la pena continuar?

Para poder tomar estas decisiones, necesitaremos haber explorado previamente cuál será la funcionalidad que hay que desarrollar, a quién puede afectar el nuevo sistema y quién tendrá que interactuar con él. También necesitare-

mos tener una idea aproximada de cuál será la solución final y su viabilidad. Finalmente, habrá que tener una estimación inicial del coste y del calendario, así como también de los riesgos que pueden afectar al desarrollo correcto del proyecto.

El segundo hito (de arquitectura) se alcanza cuando ya tenemos una versión más o menos estable y final de la arquitectura, ya hemos implementado las funcionalidades más importantes y hemos eliminado los principales riesgos. Es el momento de plantearnos si la arquitectura que hemos definido nos servirá para llevar a cabo todo el proyecto, si consideramos que los riesgos que quedan están bajo control y si creemos que estamos añadiendo valor a buen ritmo.

Para ello, habremos tenido que explorar los requisitos con más detalle, de manera que tengamos una estimación clara del coste pendiente del proyecto y de los problemas que nos podemos encontrar. También necesitaremos haber implementado un número significativo de funcionalidades para poder estar seguros de la idoneidad de la arquitectura elegida.

El tercer hito (capacidad operacional inicial) llegará en el momento en el que consideremos que el sistema está bastante desarrollado para entregarlo al equipo de transición. La principal pregunta que nos debemos plantear llegados a este punto es si podemos decir que ya hemos implementado toda la funcionalidad que había que implementar.

El último objetivo (entrega del producto) es cuando debemos decidir si hemos conseguido los objetivos iniciales del proyecto. En este caso, es importante que los clientes acepten el proyecto que hemos entregado.

Con el proceso unificado, cada una de las fases incluye actividades que alcanzan todo el ciclo de desarrollo: desde los requisitos hasta la implementación y las pruebas. Por ejemplo, en la fase de inicio, la definición de la arquitectura incluye la creación de una versión ejecutable de ésta.

La ventaja principal de este cambio de filosofía respecto al ciclo de vida en cascada es que, con el proceso unificado, podemos tomar las decisiones basándonos en información empírica y no sólo en información teórica.

Desgraciadamente, es muy habitual ver equipos que creen estar siguiendo el proceso unificado pero que en realidad están siguiendo una variante del ciclo de vida en cascada, dado que, a priori, podría parecer que las fases de UP no son muy diferentes de las etapas definidas, por ejemplo, en Métrica 3 (*inception* podría ser el PSI, *elaboration* sería EVA + ASI, *construction* DSI + CSI y *transition* IAS + MSI), pero, como hemos dicho, las diferencias son muy grandes y muy importantes: aunque los objetivos son similares, la manera de lograrlos es muy diferente.

Actividades

En cuanto a las actividades, UP define las actividades principales siguientes:

- **Modelización de negocio (*business modelling*)**. Esta actividad intenta solucionar el problema de comunicación entre los expertos en el dominio y los especialistas en tecnología que, habitualmente, usan lenguajes diferentes. Se generan los casos de uso "de negocio" (*business use cases*), que describen los procesos principales de la organización y que servirán como lenguaje común para todos los implicados en el proceso de desarrollo. Esta actividad es muy importante, ya que es la que nos asegurará que los desarrolladores entiendan cuál es la ensambladura del producto que están desarrollando dentro del contexto de la organización para la que lo están desarrollando.

- **Requisitos (*requirements*)**. Describir **qué** debe hacer el sistema (y que no debe hacer). El modelo que se usa para describir la funcionalidad del sistema es el denominado *modelo de casos de uso*, que consiste en describir escenarios de uso del sistema mediante una secuencia de acontecimientos (el usuario hace X, el sistema hace Y, etc.).

- **Análisis y diseño (*analysis* & *design*)**. Describir **cómo** se implementará el sistema. Se crean modelos detallados de los componentes que formarán el sistema. Estos modelos sirven como guía para los implementadores a la hora de escribir el código fuente de los programas. Estos modelos han de estar relacionados con los casos de uso y deben permitir introducir cambios en el supuesto de que los requisitos funcionales cambien.

- **Implementación (*implementation*)**. Definir la organización del código, escribirlo y verificar que cada componente cumple los requisitos de manera unitaria (es decir, de manera aislada del resto de los componentes) y generar un sistema ejecutable. El proceso también prevé la reutilización de componentes que existan previamente.

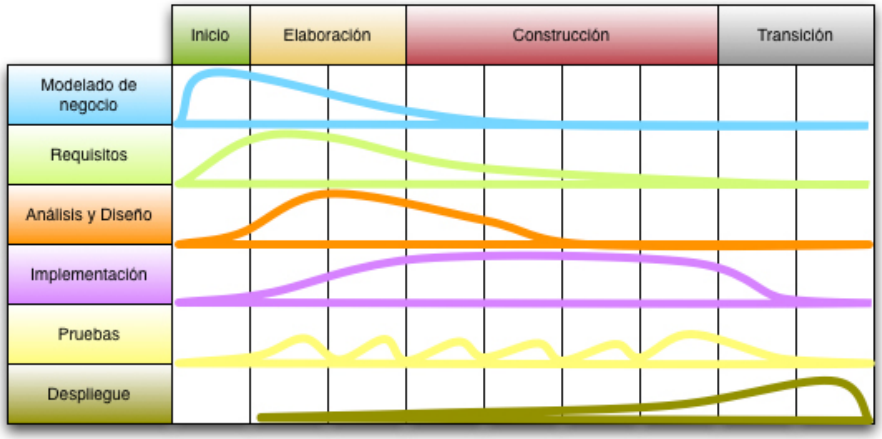
- **Pruebas (*test*)**. Verificar la interacción entre los objetos y componentes que forman el sistema. Dado que las pruebas se ejecutan durante todas las iteraciones, es más fácil detectar errores antes de que se propaguen por el sistema. Las pruebas han de incluir la fiabilidad, la funcionalidad y el rendimiento.

- ***Deployment***. Producir las entregas del sistema y entregarlas a los usuarios finales. Incluye las tareas relativas a la gestión de la configuración y de las versiones y, sobre todo, tiene lugar durante la fase de transición.

Además de las actividades aquí descritas, el proceso unificado también prevé otras actividades, como la personalización del propio proceso (recordemos que UP es un bastimento de procesos y no un proceso concreto), la gestión de la configuración y del cambio y también la planificación y gestión del proyecto.

Es importante recordar que todas las actividades tienen lugar durante todas las etapas, aunque con diferente énfasis. Así, durante la etapa de elaboración, se dedicará más tiempo a la modelización de negocio que a la implementación, mientras que durante la etapa de construcción, la proporción se verá invertida. Esta variabilidad en el énfasis que se dedica a cada actividad se suele representar mediante el diagrama siguiente.

Énfasis de las diferentes actividades en las fases del proceso unificado



Roles

OpenUP define los roles siguientes:

- ***Stakeholder***. Cualquier persona que tenga un interés en el resultado final del proyecto.

- **Jefe de proyecto**. Encargado de la planificación y de coordinar la interacción entre los *stakeholders*. También es responsable de mantener a los miembros del equipo centrados en conseguir cumplir los objetivos del proyecto.

- **Analista**. Recoge la información de los *stakeholders* a modo de requisitos, los clasifica y los prioriza. Es el que, en la clasificación que hemos hecho en el subapartado 2.4, hemos denominado *analista funcional*.

- **Arquitecto**. Define la arquitectura del software, lo que incluye tomar las decisiones clave que afectan a todo el diseño y a la implementación del sistema.

- **Desarrollador**. Desarrolla una parte del sistema, lo que incluye diseñarla de acuerdo con la arquitectura, prototipar (si es necesario) la interfaz gráfica de usuario, implementarla y probarla tanto aislada del resto del siste-

ma como integrada con el resto de los componentes. Este rol incluye los roles de analista orgánico y programador, tal y como los hemos definido en el subapartado 2.4.

- **Experto en pruebas**. Identifica, define, implementa y lleva a cabo las pruebas aportando un punto de vista complementario al del desarrollador. Lo más habitual es que trabaje sobre el sistema construido y no sobre componentes aislados.

3.3.3. Scrum

A pesar de ser contemporáneo de UP, Scrum se popularizó posteriormente como parte del movimiento del desarrollo ágil. Es un método iterativo e incremental para desarrollar software elaborado por Ken Schwaber y Jeff Shuterland. Se trata de un método muy ligero que, siguiendo el principio *lean* de generar sólo los artefactos que aportan un valor importante, minimiza el conjunto de prácticas y artefactos, así como también las tareas y los roles.

Web recomendada

La descripción del método Scrum la encontraréis en *Scrum Guide*: http://www.scrumguides.org/scrum-guide.html (última visita: septiembre 2010).

Roles

Scrum distingue, de entrada, entre dos grandes grupos de participantes de un proyecto: los que están comprometidos en el desarrollo (el equipo) y los que están involucrados pero no forman parte del equipo (lo que otros métodos denominan *stakeholders*).

Pollos y cerdos

Scrum denomina *pollos* a los *stakeholders* y *cerdos* al equipo. Estos nombres provienen de la siguiente historia: un pollo y un cerdo se encuentran y el pollo le pregunta: ¿montamos un restaurante?. El cerdo se lo piensa y le pregunta a su vez: ¿cómo lo llamaríamos?. El pollo contesta: huevos con tocino. A lo que el cerdo responde: No, gracias, porque yo estaría comprometido y tú sólo estarías involucrado.

Entre los miembros del equipo, Scrum define tres roles muy particulares:

- ***Scrum Master***. Es responsable de asegurar que el equipo sigue los valores, las prácticas y las reglas de Scrum. También se encarga de eliminar los impedimentos que el equipo se pueda encontrar dentro de la organización.

- ***Product owner***. Es la persona responsable de velar por los intereses de todos los *stakeholders* y llevar el proyecto a buen término. Es, por lo tanto, el encargado de decidir qué se implementa y qué es más prioritario.

- ***Team***. El resto de los miembros del equipo son los desarrolladores. No se especializan, sino que todos deben tener, en mayor o menor medida, habilidades en todas las actividades implicadas. Los miembros del equipo deciden ellos mismos cómo se organizan el trabajo y nadie (ni siquiera el *Scrum Master*) les puede decir cómo lo deben hacer.

Artefactos

Scrum sólo define cuatro artefactos:

- ***Product backlog***. Es la lista de requisitos pendientes de implementar en el producto. Cada entrada (normalmente una "historia de usuario") tiene asociada una estimación del valor que aporta a la organización y también del coste de su desarrollo.

- ***Sprint backlog***. El *backlog* para una iteración concreta (Scrum denomina *sprint* a las iteraciones); más detallada que el *product backlog* y en la que cada historia de usuario se descompone en tareas de entre cuatro y dieciséis horas de duración.

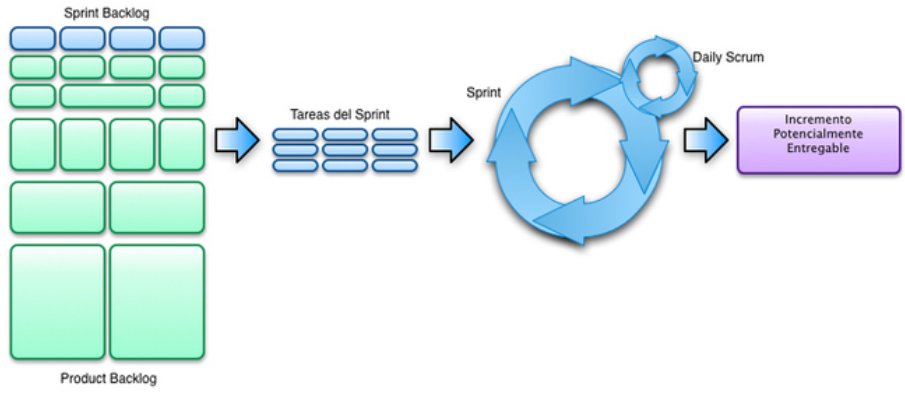
- ***Release burndown chart***. Gráfico que muestra el progreso actual del equipo en función del número de historias de usuario que faltan por implementar.

- ***Sprint burndown***. El *burndown* para una iteración concreta, en la que el progreso se puede medir en tareas finalizadas aunque no sean historias de usuario completas.

Prácticas

En cuanto a las prácticas, éstas se basan en dos ciclos de iteraciones: una iteración más larga (como las que podemos encontrar en métodos como UP) y una "iteración diaria".

Proceso de Scrum



- ***Sprint planning meeting***. Reunión que se organiza antes de empezar un *sprint* y en la que se deciden qué historias de usuario se implementarán en este *sprint*. Por lo tanto, se crea el *sprint backlog*.

- ***Daily scrum***. Reunión diaria en la que todos los miembros del equipo responden a tres preguntas: qué hicieron ayer, qué piensan hacer hoy y qué impedimentos se han encontrado que les han impedido avanzar. La finalidad de esta reunión es que todos los miembros del equipo estén enterados

de lo que está haciendo el resto de los miembros y que así se identifiquen oportunidades de ayudarse unos a otros.

- ***Sprint review meeting***. Al finalizar un *sprint* se revisa el trabajo realizado y se enseña a quien esté interesado el resultado implementado (la *demo*).

- ***Sprint retrospective***. Sirve para reflexionar sobre lo que haya pasado durante el *sprint* y para identificar oportunidades de mejora en el proceso de desarrollo.

4. Técnicas y herramientas de la ingeniería del software

La ingeniería del software no consiste sólo en una manera de organizar el desarrollo, también nos proporciona un conjunto de técnicas y herramientas que nos deben ayudar a poner en práctica los métodos que hayamos elegido para desarrollar nuestro proyecto de software.

De todo el abanico de técnicas de la ingeniería del software, veremos sólo unas cuantas, que hemos separado en dos grandes grupos: las técnicas basadas en la reutilización y las basadas en la abstracción.

También hablaremos de las herramientas de apoyo a la ingeniería del software, ya que estas herramientas nos ayudan a aplicar con éxito las técnicas mencionadas.

4.1. Técnicas basadas en la reutilización

Uno de los principales objetivos de la ingeniería del software es favorecer la reutilización. Al desarrollar un nuevo producto, la situación ideal es no partir de cero, sino tener partes ya desarrolladas y aprovecharlas en el nuevo contexto. Esto nos aportará una serie de ventajas:

Lectura recomendada

Bertrand Meyer (1999). *Construcción de software orientado a objetos*. Madrid: Prentice Hall.

- **Oportunidad**. Dado que debemos desarrollar menos software, éste se puede desarrollar con más rapidez y, por lo tanto, tenerlo terminado antes.

- **Disminución de los costes**. Puesto que el componente es compartido, el coste de su desarrollo y mantenimiento también puede ser compartido.

- **Fiabilidad**. Los componentes reutilizados ya han sido probados y utilizados y, por lo tanto, podemos confiar en su buen funcionamiento, dado que, si tienen errores, algún otro los puede haber encontrado antes.

- **Eficiencia**. El desarrollador del componente reutilizable se puede especializar en un problema muy concreto (el que resuelve el componente) y, por lo tanto, encontrar las soluciones más eficientes.

Tal y como dice Meyer (1999), la reutilización tiene una gran influencia sobre todos los otros aspectos de la calidad del software, ya que al resolver el problema de la reutilización se deberá escribir menos software y, en consecuencia, se podrán dedicar más esfuerzos (por el mismo coste total) a mejorar los otros factores, como la corrección y la robustez.

Ejemplo de reutilización

Un ejemplo sencillo de reutilización son los controladores de los dispositivos. En los tiempos del MS-DOS (años ochenta-comienzos de los noventa) los desarrolladores de aplicaciones debían escribir sus propios controladores de los dispositivos (por ejemplo, de la impresora o de la tarjeta de sonido) y, por lo tanto, debían dedicar parte de los recursos a soportar una gama lo más amplia posible de dispositivos. Uno de los motivos del éxito del sistema Windows fue, precisamente, la incorporación de un sistema de controladores genérico que permitió a los desarrolladores centrarse en otras áreas del programa que aportaran más valor añadido.

La reutilización no es en absoluto un concepto nuevo, sino que ha estado presente a lo largo de la historia del desarrollo y, de hecho, es anterior a la ingeniería del software. A lo largo del tiempo se ha ido aumentando progresivamente el nivel de reutilización, pero como el software ha ido ganando en complejidad durante este periodo (en parte, gracias a la reutilización), la necesidad de reutilización, en vez de disminuir, ha crecido.

Hoy en día, ejemplos como el que hemos comentado anteriormente ya no se consideran ejemplos de reutilización, sino que se dan por supuestos y de los ingenieros del software se espera que logren nuevos hitos en este sentido y reutilicen componentes cada vez mayores y más complejos, así como que apliquen la reutilización a todas las actividades posibles: desde la adquisición de requisitos hasta la implementación y el despliegue.

Sin embargo, la reutilización no está exenta de costes y problemas. Para favorecer la reutilización, hay que desarrollar los diferentes componentes del software, dado que, en el caso de no hacerlo, será muy difícil aprovecharlos en un contexto diferente del contexto en el que se crearon.

Además, deberemos documentar los diferentes componentes de manera mucho más exhaustiva, ya que, idealmente, los equipos que reutilicen un componente deberían poder hacerlo sin necesidad de ver ni modificar su código fuente.

Por otro lado, habrá que ir con especial cuidado a la hora de desarrollar y mantener un componente reutilizado, dado que cualquier defecto que éste tenga se propagará en todos aquellos sistemas que lo usan.

Finalmente, también habrá que llevar a cabo una gestión exhaustiva de las diferentes versiones del componente y, a la hora de evolucionarlo, valorar todos los contextos en los que se está usando. Un mismo componente puede

ser utilizado en varios sistemas y, seguramente, cada sistema usará una versión diferente. Cuando haya que corregir errores o añadir funcionalidad, habrá que considerar con atención la compatibilidad entre versiones.

Imaginemos que desarrollamos un componente C para un determinado sistema S1. Al desarrollar un segundo sistema S2 nos damos cuenta de que podemos reutilizar el componente C y entonces lo hacemos. Para ello debemos registrar que la versión actual del componente es la 1.0 y que está en uso en los sistemas S1 y S2.

Más adelante, el cliente nos pide un cambio en el sistema S1 que, al implementarlo, afecta al componente C y origina la versión 2.0, incompatible con la 1.0. Ahora tenemos el sistema S1, que usa la versión 2.0, y el sistema S2, con la 1.0.

Si, posteriormente, detectamos un error en el componente C que afecta a la versión 1.0, deberemos corregirlo en esta versión (y crear, por ejemplo, la versión 1.1, que usará el sistema S2), pero también en la versión 2.0 (y crear así la versión 2.1, que se usará en el sistema S1).

En una escala no tecnológica, la reutilización también genera cuestiones como, por ejemplo, el modo como podemos asignar el coste del desarrollo de un componente reutilizable, qué pasa si después nadie lo reutiliza o quién se debe encargar de su mantenimiento.

Una posibilidad es el desarrollo, a priori, de componentes reutilizables, pero como dice Meyer (1999) "debemos encontrar primero las maneras de resolver un problema antes de preocuparnos por aplicar la solución a otros problemas".

Dicho de otro modo: es muy difícil crear una solución genérica si no se ha creado antes una solución específica que nos permita explorar todos los aspectos importantes del problema.

Finalmente, la estandarización de los componentes en la industria requiere la elaboración de una serie de normas y acuerdos que deben ser aceptados por todos los participantes, y esto no es siempre fácil.

La historia de la reutilización ha sido bastante complicada y no han faltado los éxitos ni los fracasos. Sin embargo, hoy en día aún se continúa investigando en nuevas maneras de incrementar el grado de reutilización y llevarlo al nivel que poseen otras ingenierías, en las que el grado de estandarización y reutilización de los componentes es todavía bastante más alto que en el caso de la ingeniería del software.

A continuación, estudiaremos algunas de las técnicas que se han ido desarrollando a lo largo de los años para mejorar los niveles de reutilización.

4.1.1. Desarrollo orientado a objetos

La programación orientada a objetos se basa en dos técnicas básicas que facilitan la reutilización: la ocultación de información y la abstracción.

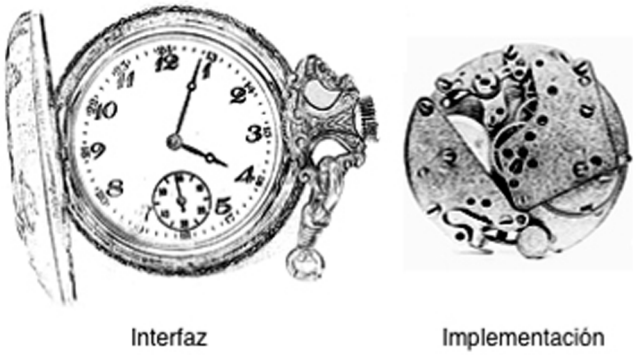
Ver también

Hablaremos de la programación orientada a objetos con más detalle en el módulo "Orientación a objetos".

La ocultación de información consiste en esconder los detalles sobre la estructura interna de un módulo, de manera que podemos definir claramente qué aspectos de los objetos son visibles públicamente (y, por lo tanto, utilizables para los reutilizadores) y cuáles no.

Denominamos *interfaz* a aquella parte de un componente de software que es visible a los programadores que lo usan. Así, cuando un programador reutiliza un objeto, sólo debe conocer la interfaz y, en todo caso, un contrato que le indique cómo funciona la interfaz. Por oposición, la implementación de un componente es aquella parte del componente que queda oculta a sus usuarios.

Interfaz e implementación



En esta figura, hemos distinguido la interfaz del componente reloj de su implementación. Como usuarios, nos basta con saber que el reloj nos ofrece una única operación (¿qué hora es?) y que nos da esta información mediante las manecillas. Esto permite que usemos el reloj con independencia del mecanismo (a cuerda, a pila, etc.) con el que esté implementado.

La ocultación de información facilita la reutilización porque si un mantenimiento correctivo o evolutivo de un componente se puede hacer modificando sólo la implementación, sin modificar la interfaz, entonces los usuarios de este componente no se verán afectados.

La abstracción consiste en identificar los aspectos relevantes de un problema, de manera que nos podamos concentrar sólo en éstos.

En el caso de la orientación a objetos, la abstracción nos ayuda a agrupar los aspectos comunes a una serie de objetos, de manera que sólo los debemos definir una vez (es lo que se denomina *clase*) y a crear nuevas clases de objetos e indicar sólo aquellas características que les sean específicas.

Sin embargo, la manera habitual de reutilizar clases en orientación a objetos no es reutilizando clases aisladas, sino creando bibliotecas de clases especializadas que después los desarrolladores pueden usar sin necesidad de conocer cómo están implementadas.

Ejemplos de reutilización mediante orientación a objetos

Uno de los motivos del éxito del lenguaje Java en entornos empresariales es la extensa biblioteca de clases que incorpora el lenguaje. Los programadores Java, por ejemplo, pueden reutilizar clases que implementan conceptos matemáticos, estructuras de datos, fechas con varios sistemas de calendario o algoritmos de criptografía, por lo que se pueden centrar en desarrollar lo que es específico de su aplicación.

Otro gran ejemplo de reutilización mediante orientación a objetos es la biblioteca de clases Standard Template Library. Se trata de una biblioteca de clases para el lenguaje C++, estandarizada en 1994, que facilita el trabajo con colecciones de objetos.

4.1.2. Bastimentos

Un bastimento es un conjunto de clases predefinidas que debemos especializar para implementar una aplicación o subsistema. Del mismo modo que las bibliotecas de clases, el bastimento nos ofrece una funcionalidad genérica ya implementada y probada que nos permite centrarnos en aquello que es específico de nuestra aplicación.

Sin embargo, a diferencia de las bibliotecas de clases, un bastimento define el diseño de la aplicación que lo usará. Es decir, no solamente estamos reutilizando la implementación de las clases, sino que de manera indirecta también estamos reutilizando el diseño del sistema. Normalmente, son las clases del bastimento las que llamarán a nuestras clases y no a la inversa. Por lo tanto, el nivel de dependencia de nuestro sistema respecto al bastimento es más alto que en el caso de usar una biblioteca de clases.

En cambio, un buen bastimento nos ofrece un nivel de reutilización más alto y, normalmente, nos facilita el seguimiento de una serie de prácticas y principios ya probados, con lo que mejora significativamente la calidad de la solución final.

Ejemplo de bastimento

Un ejemplo de bastimento serían los bastimentos para el desarrollo de aplicaciones web, como Ruby on Rails. Este bastimento se encarga de numerosas tareas de bajo nivel para que, cuando un usuario visita una determinada página web, el bastimento acabe llamando a un cierto código y genere la página web de manera dinámica. El bastimento no sólo nos permite reutilizar clases, sino que también nos ayuda a reutilizar el diseño completo del sistema final. Por otro lado, a la hora de llevar a cabo tareas complejas, el programador emplea las clases del bastimento y éstas realizan llamamientos a las del programador.

4.1.3. Componentes

Las técnicas vistas hasta ahora sólo nos permiten reutilizar clases individuales o, en todo caso, agrupaciones de clases a modo de bibliotecas y bastimentos. Sin embargo, a menudo queremos aprovechar más funcionalidad de la que nos

da una sola clase. Por ello, surgió la programación orientada a componentes: con la programación orientada a componentes, la unidad de reutilización (el componente) puede estar formada por una o más clases.

A diferencia de una biblioteca de clases, el conjunto de clases que forma un componente se reutiliza como un todo; no podemos reutilizar, por lo tanto, una de las clases del componente de manera individual.

Una de las ventajas de los componentes es que están desarrollados con la intención de ser reutilizables y, por lo tanto, suelen ser más estrictas en cuanto a la ocultación de información: cada componente representa una unidad con una interfaz y un contrato que indica cómo se puede utilizar, pero que no da ningún detalle sobre cómo está desarrollado por dentro.

Interfaces gráficas de usuario

Un caso de éxito de los componentes es el de las interfaces gráficas de usuario. En entornos como .net y Java no sólo tenemos disponibles un gran número de componente de serie, sino que además existe un gran número de compañías que se dedican al desarrollo de componentes de interfaz gráfica de usuario compatibles con la plataforma. De este modo, si, por ejemplo, queremos crear un gráfico avanzado a partir de unos datos y los componentes de serie no son suficientemente sofisticados, podemos comprar un componente de gráficos a un tercero y ahorrarnos el desarrollo.

Para garantizar el éxito de un sistema basado en componentes, es importante definir claramente el componente: cuál es su interfaz (qué operaciones se pueden invocar y con qué parámetros) y cuál es su contrato. Éste nos indica cuáles son los efectos de invocar una operación (también denominados *poscondiciones*) y cuáles son las condiciones que hay que verificar para evitar que se produzca un error (también denominadas *precondiciones*).

En algunos sistemas, la interfaz se define en un lenguaje específico para las interfaces (es lo que se denomina IDL[8]), de manera que se facilita la utilización del componente aunque estemos implementando nuestro sistema en un lenguaje de programación diferente del lenguaje en el que se programó el componente.

[8]IDL son las siglas de *interface definition language*.

El desarrollo de componentes reutilizables es un poco diferente del desarrollo de componentes no reutilizables, en el sentido de que tenemos mucho menos control sobre las condiciones en las que se utilizará el componente y, por lo tanto, es más importante asegurar la calidad: por un lado, el componente debe soportar el mal uso sin que éste tenga consecuencias graves y, por el otro, debemos evitar al máximo los errores, dado que se contagiarían a los sistemas en los que se usan.

4.1.4. Desarrollo orientado a servicios

La arquitectura orientada a servicios tiene como objetivo incrementar el nivel de granularidad de la unidad de abstracción: en lugar de reutilizar una parte del código del sistema que queremos desarrollar, lo que proponen las arquitecturas orientadas a servicios es reutilizar el servicio completo.

Una aplicación con arquitectura orientada a servicios, por lo tanto, estará formada por una serie de servicios, que son unidades aisladas entre sí que colaboran a través de una red (y normalmente de manera remota) para implementar la funcionalidad del sistema.

A diferencia de los componentes, que formarán parte de nuestro sistema y que, por lo tanto, se desplegarán junto con este, los servicios tienen su ciclo de vida independiente propio y, por lo tanto, se desarrollan y se despliegan de manera separada de sus clientes.

Pago con tarjeta de crédito

Muchas entidades bancarias ofrecen, por ejemplo, la posibilidad de pagar con tarjeta de crédito desde las aplicaciones desarrolladas por terceras organizaciones. A veces, esta posibilidad se ofrece como un componente (que hay que incluir en el software desarrollado para poder efectuar el pago), pero a menudo se trata de un servicio. En este caso, el servicio se ejecuta en uno de los ordenadores de la entidad bancaria y ésta se encarga de su ciclo de vida (desarrollar y desplegar nuevas versiones para corregir errores o añadir funcionalidad); el software que lo usa, en lugar de incluir un componente que puede efectuar el pago, realiza llamadas remotas al servicio de la entidad bancaria para pedir las operaciones necesarias para ejecutar este mismo pago.

4.1.5. Patrones

Hasta ahora hemos visto maneras de reutilizar la implementación. El problema de las técnicas de reutilización de la implementación es que, muchas veces, a pesar de haber ciertos paralelismos entre las diferentes situaciones, no podemos aplicar exactamente la misma solución a dos contextos diferentes.

Por ejemplo, si estuviéramos desarrollando una aplicación con una base de datos remota, nos encontraríamos con el problema de que la conexión a la base de datos puede estar abierta o cerrada y que, por lo tanto, debemos lograr que la conexión se comporte de manera diferente según esté abierta o cerrada. Por otro lado, en la interfaz gráfica, nos encontramos con que ciertos elementos pueden estar activos o inactivos y que, de nuevo, se comportarán de manera diferente según estén o no activos.

En estos dos casos, a pesar de haber un cierto paralelismo (un elemento del programa se comporta de manera diferente según su estado), no podemos crear una clase genérica que podamos reutilizar en los dos casos, como tampoco podemos crear un componente que nos implemente este comportamiento.

Necesitamos, pues, herramientas de reutilización que nos permitan reutilizar ideas de manera independiente de la implementación: es como actúan los patrones.

El uso de patrones se originó en el mundo de la arquitectura y el urbanismo a finales de los años setenta. Desde entonces muchos autores han trabajado en la manera de trasladar esta práctica al desarrollo del software, pero fue a mediados de los años noventa cuando se publicó el libro que popularizó definitivamente el uso de patrones: *Design Patterns*: *Elements of Reusable Object-Oriented Software*, por parte de Erich Gama, Richard Helm, Ralph Johnson y John Vlissides.

> Erich Gama, Richard Helm, Ralph Johnson y John Vlissides definen un patrón de diseño de la manera siguiente: "un patrón de diseño da nombre, motiva y explica de manera sistemática un diseño general que permite solucionar un problema recurrente de diseño en sistemas orientados a objetos. El patrón describe el problema, la solución, cuándo se debe aplicar la solución y también sus consecuencias. También da algunas pistas sobre cómo se debe implementar y ejemplos [...]. La solución se ha de personalizar e implementar para solucionar el problema en un contexto determinado".

A partir de aquí, el uso de patrones se fue extendiendo hacia otras actividades, como el análisis o la arquitectura del software, ya que son una herramienta muy útil para reutilizar ideas en estos contextos. Las principales ventajas de los patrones son:

- Reutilizar las soluciones y aprovechar la experiencia previa de otras personas que han dedicado más esfuerzo a entender los contextos, las soluciones y las consecuencias de lo que nosotros queremos o podemos dedicar.
- Beneficiarnos del conocimiento y la experiencia de estas personas mediante un enfoque metódico.
- Comunicar y transmitir nuestra experiencia a otras personas (si definimos otros nuevos).
- Establecer un vocabulario común para mejorar y agilizar la comunicación.
- Encapsular conocimiento detallado sobre un tipo de problema y sus soluciones, asignándole un nombre para que podamos hacer referencia a él fácilmente.
- No tener la necesidad de reinventar una solución al problema.

Para conseguir estos beneficios, deberemos documentar, como mínimo, los elementos del patrón siguientes:

- El nombre nos permite hacer referencia al patrón cuando documentamos nuestro diseño o cuando lo comentamos con otros desarrolladores. También nos permite aumentar el nivel de abstracción del proceso de diseño, dado que podemos pensar en la solución al problema como un todo.

- El problema nos indica qué resuelve el patrón. Este problema podría ser un problema concreto o la identificación de una estructura problemática (por ejemplo, una estructura de clases poco flexible).

- La solución describe los elementos que forman parte del patrón, así como sus relaciones, responsabilidades y colaboraciones. La solución no es una estructura concreta, sino que, como hemos indicado anteriormente, actúa como una plantilla que podemos aplicar a nuestro problema. Algunos patrones presentan varias variantes de una misma solución.

- Las consecuencias son los resultados y los compromisos derivados de la aplicación del patrón. Es muy importante que las tengamos en cuenta a la hora de evaluar nuestro diseño, dado que es lo que nos permite entender realmente el coste y los beneficios de la aplicación del patrón.

Patrón *estado*

Por ejemplo, el patrón *estado* nos puede ayudar a solucionar el problema mencionado anteriormente de clases que se comportan de manera diferente en función de su estado. La definición (simplificada) que dan Gama, Helm, Johnson y Vlissides (1994) del patrón *estado* es la siguiente:

- **Nombre**: estado (también conocido como *objetos por estados*).

- **Problema**: el comportamiento de un objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución en función de éste. Las operaciones tienen sentencias condicionales largas que dependen del estado del objeto. A menudo varias operaciones tienen las mismas estructuras condicionales (es decir, prevén las mismas condiciones, dado que dependen de los posibles estados del objeto).

- **Solución**: introducir una nueva clase que representa el estado con una subclase para cada estado que tenga un comportamiento diferente. Cada rama de las estructuras condicionales se corresponderá ahora con una de las subclases del estado. Esto nos permite tratar el estado del objeto como un objeto en sí mismo que puede cambiar independientemente del resto de los objetos.

- **Consecuencias**: localiza el comportamiento específico del estado, divide el comportamiento de los diferentes estados y hace explícitas las transiciones entre estados.

Observación

Todavía no hemos explicado algunos de los conceptos que se utilizan en la descripción de un patrón (tened en cuenta que se trata de un patrón de diseño de software), pero esto no nos debe preocupar, puesto que, ahora mismo, lo que nos interesa es ver qué estructura tiene un patrón y hacernos una idea aproximada de cómo lo usaríamos.

4.1.6. Líneas de producto

Las líneas de producto son, en origen, una estrategia de marketing que consiste en ofrecer, de manera individual, una serie de productos relacionados entre sí. La línea de producto permite mantener una identidad base común a todos los productos de la línea y, al mismo tiempo, ofrecer un producto adaptado a los gustos del consumidor.

Un ejemplo típico de líneas de producto es el caso de la automoción, en el que podemos elegir, para un determinado modelo, diferentes niveles de acabados del interior, de color, motor, carrocería, etc.

En industrias como la de la automoción, una de las ventajas de las líneas de producto es que facilitan la reutilización de componentes comunes entre los diferentes productos de la misma línea, razón por la que no han faltado esfuerzos para trasladar este concepto al mundo del software.

Líneas de productos de software

De hecho, podemos considerar que, actualmente, existen muchas líneas de productos de software. Por ejemplo, de la distribución de Linux Ubuntu se hallan, en el año 2010, tres ediciones oficiales (Desktop, Server y Netbook) e infinidad que derivan de éstas, todas basadas en el núcleo de la distribución Debian, pero configurando paquetes de software diferentes (entorno de escritorio, navegador, utilidades, etc.) para personalizar el producto según el mercado al que se dirija la distribución.

Desde un punto de vista más relacionado con el desarrollo del software, hay técnicas y tecnologías específicas que facilitan la creación de líneas de productos de software, a pesar de que su estudio queda fuera del ámbito de estos materiales.

4.2. Técnicas basadas en la abstracción

Otro de los retos de la ingeniería del software es facilitar la creación de abstracciones que permitan desarrollar sistemas usando lenguajes más cercanos a los lenguajes naturales que a los lenguajes de los ordenadores.

De hecho, la mayoría de las técnicas de reutilización que acabamos de ver se basan en la creación de abstracciones. También hemos visto como, a lo largo del tiempo, se han ido creando lenguajes cada vez más sofisticados para comunicarnos con los ordenadores: desde los lenguajes ensamblador y el código máquina hasta los lenguajes visuales de modelización.

Estos nuevos lenguajes tiene en común la característica de que siempre sabemos traducirlos a otros lenguajes "de menor nivel" y así sucesivamente hasta llegar al lenguaje máquina, ya sea intermediando un compilador (que es un programa que genera un programa en un lenguaje destino a partir de un programa en un lenguaje origen) o un intérprete (que es un programa que lee, interpreta y ejecuta un programa escrito en un cierto lenguaje).

Tecnología JSP

Por ejemplo, la tecnología de páginas de servidor JSP nos permite escribir un programa en un lenguaje similar al HTML (el lenguaje JSP), que entonces será traducido al lenguaje Java, desde el que será traducido a un lenguaje denominado Java Bytecode, que será interpretado y ejecutado. Todas estas traducciones tienen lugar de manera totalmente automatizada y transparente al programador de la página JSP.

Denominamos *ingeniería dirigida por modelos*[9] a un conjunto de técnicas basadas en la abstracción que ven el desarrollo de software como una actividad en la que, principalmente, se desarrollan modelos (en el lenguaje que sea) y que son el artefacto principal, con independencia de la existencia o no de código fuente (sea generado automática o manualmente).

(9)En inglés, *model-driven engineering* (MDE).

A continuación, veremos dos técnicas de ingeniería dirigida por modelos: la arquitectura dirigida por modelos y los lenguajes específicos del dominio.

4.2.1. Arquitectura dirigida por modelos

El término *arquitectura dirigida por modelos*[10] hace referencia a un estándar publicado por el OMG cuyo objetivo es separar la lógica de negocio y de aplicación (es decir, la funcionalidad del sistema por desarrollar) de la tecnología en la que se implementará el sistema.

(10)En inglés, *model driven architecture* (MDA).

Enlace de interés

Página oficial del OMG sobre MDA:
http://www.omg.org/mda

En MDA, un modelo es una descripción o especificación de un sistema y de su entorno para un cierto propósito. Normalmente, el modelo se presentará como una combinación de notación gráfica y textual. El texto podría estar en un lenguaje de modelización o en lenguaje natural.

La filosofía de MDA consiste en definir, además de los modelos, una serie de reglas de transformación entre modelos que nos permitan generar, de la manera más automatizada posible, programas a partir de los modelos, lo que nos permitirá ahorrarnos la fase de implementación.

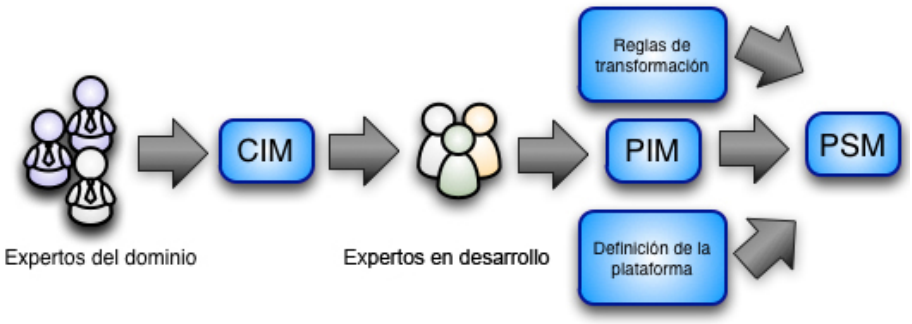
El punto de partida en un desarrollo MDA es un modelo que denominamos *modelo independiente de la computación*[11], que es el que sirve para comunicar los requisitos de los expertos en el dominio a los expertos en diseño y construcción de software (los desarrolladores). Este modelo también se conoce como modelo del dominio o modelo del negocio.

(11)En inglés, *computation independient model* (CIM)

El CIM no tiene en cuenta la tecnología con la que se implementará el sistema y, por lo tanto, no está bastante detallado para ser ejecutado directamente. Por ello, el CIM debe ser transformado en un segundo modelo, más cercano a la tecnología, denominado *platform independient model* (PIM).

El PIM, a pesar de estar más próximo a la tecnología, es independiente de la plataforma final de ejecución, en el sentido de que un mismo PIM puede dar lugar a diferentes implementaciones sobre distintas plataformas. Por ejemplo, un mismo PIM podría dar lugar a una implementación para la plataforma Java o a una implementación para la plataforma Microsoft.Net. De este modo, conseguimos uno de los objetivos de MDA: separar la evolución de los modelos de la evolución de la tecnología.

Acompañando al PIM tendremos un conjunto de reglas de transformación hacia diferentes plataformas. Estas reglas nos servirán para conseguir el denominado *platform specific model* (PSM), que puede ser compilado y ejecutado directamente sobre la plataforma.



Sin embargo, MDA también prevé otras posibilidades, como generar directamente código ejecutable a partir del PIM o generar un segundo PSM a partir de un PSM inicial para añadir nuevos detalles de implementación. Lo importante, pues, es el proceso de refinamiento sucesivo de modelos.

4.2.2. Lenguajes específicos del dominio

Mientras que MDA está históricamente ligado al uso de modelos gráficos, también podríamos crear nuestros modelos mediante lenguajes de cariz más textual, que denominamos *lenguajes específicos del dominio* (DSL, *domain-specific languages*).

Lectura recomendada

Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?

El desarrollo basado en DSL consiste en crear lenguajes nuevos adaptados al dominio del sistema que queremos desarrollar, de manera que estén optimizados para expresar los requisitos del sistema que estamos desarrollando.

A diferencia, pues, del lenguaje utilizado en MDA (normalmente, UML), que es un lenguaje de propósito general, los lenguajes específicos del dominio sólo sirven para solucionar un problema muy concreto (entender un formato de intercambio de datos, describir un proceso de negocio, etc.). Esto facilita que sean muy sencillos pero, al mismo tiempo, obliga a definir más de uno para cubrir la funcionalidad completa de un sistema.

4.3. Herramientas de apoyo a la ingeniería del software (CASE)

La ingeniería del software, como cualquier otra actividad humana, requiere unas herramientas que nos ayuden con las diferentes tareas. Cuando estas herramientas consisten en aplicaciones informáticas, se conocen como herramientas CASE[12].

(12)CASE son las siglas de *computer aided software engineering*.

La finalidad de las herramientas CASE es ayudar al ingeniero de software a aplicar los principios de la ingeniería en el desarrollo de software. Así, no todas las herramientas que se usan durante el desarrollo del software entrarían en esta categoría. Por ejemplo, las herramientas de programación no se consideran, normalmente, herramientas CASE, pero las herramientas de modelización sí.

La principal ventaja de este tipo de herramientas es que facilitan la automatización de algunas de las tareas del ingeniero de software, así como el tratamiento informático de los productos de su trabajo. Por ejemplo, una herramienta de modelización nos puede ayudar a mantener consistentes todos los modelos mediante una serie de validaciones y comprobaciones automatizadas.

Herramientas de modelización optimizadas para elaborar diagramas UML

Para ejecutar un diagrama UML podríamos usar cualquier herramienta de dibujo, pero el hecho de que las herramientas de modelización estén optimizadas para elaborar diagramas UML hace que sea mucho más cómodo y sencillo realizarlos con estas herramientas que con los programas clásicos de dibujo. Estas herramientas, además, pueden hacer validaciones sobre qué "dibujos" son modelos válidos y cuáles no tienen sentido.

Una herramienta CASE puede ser algo tan sencillo como una herramienta que nos ayude a dibujar los diagramas o algo tan complicado como una familia de herramientas que nos ayuden a seguir estrictamente un proceso determinado de desarrollo. En ambos casos, es importante que las diferentes herramientas se comuniquen entre sí.

Estas herramientas nos ayudan a gestionar el proyecto, dado que, como hemos dicho anteriormente, nos pueden ayudar a implementar un proceso de desarrollo y, por lo tanto, a hacer el seguimiento de las diferentes actividades y tareas.

Dentro del ámbito más específico de la ingeniería del software, las herramientas CASE nos pueden ayudar mucho en la gestión de los requisitos. Las herramientas de gestión de los requisitos nos ayudan a mantener un repositorio centralizado de requisitos y a consultar la historia de un requisito desde la petición del *stakeholder* hasta la implementación en código (según la herramienta).

Otra actividad que se puede beneficiar mucho del uso de una herramienta CASE es, como hemos dicho anteriormente, la modelización en particular y el análisis en general. Las herramientas CASE nos pueden ayudar, incluso, a transformar los modelos en código ejecutable (como es el caso del desarrollo

basado en modelos) o a generar modelos automáticamente a partir de código (es lo que se denomina *ingeniería inversa*). En una situación ideal, la herramienta CASE debería poder actualizar el código a partir de los modelos y viceversa (es lo que se denomina *round-trip engineering*).

Finalmente, la gestión de la calidad también se puede ver muy beneficiada por el uso de herramientas que nos generen métricas de manera automatizada o nos ayuden a gestionar los errores detectados, así como a analizar las causas para encontrar procesos de mejora que nos ayuden a reducir el volumen de errores (por ejemplo, si detectamos que el 30% de los errores se producen durante la construcción del sistema, sabemos que debemos mejorar las pruebas que llevamos a cabo de esta actividad).

Las herramientas CASE, finalmente, también nos pueden ayudar en las actividades de mantenimiento y reingeniería:

- **Herramientas de gestión del proceso**. Ayudan a la definición, modelización, ejecución o gestión del proceso del propio desarrollo.

- **Herramientas de gestión de proyectos**. Herramientas que ayudan en tareas de gestión del proyecto, como la valoración, la planificación o el análisis del riesgo.

- **Herramientas de gestión de requisitos**. Herramientas de apoyo para la recogida, documentación, gestión y validación de requisitos. Pueden ser genéricas, en las que los requisitos se documentan textualmente, o específicas, como las basadas en historias de usuario o en casos de uso.

- **Herramientas de modelización**. Herramientas que facilitan la creación de modelos. En el caso específico de UML, permiten la creación de diagramas UML y validan en mayor o menor grado la corrección de la sintaxis utilizada.

- **Herramientas de ingeniería inversa**. Se trata de herramientas que permiten analizar un software existente para poder empezar a aplicar los principios de ingeniería del software; normalmente elaboran modelos de análisis y diseño, por ejemplo, en UML, a partir del código, de una base de datos existente, etc.

- **Entorno integrado de desarrollo**. Herramientas para la codificación del software, pero también para el diseño, la ejecución de pruebas y la depuración.

- **Herramientas de construcción del software**. Herramientas que construyen el ejecutable final a partir de los varios archivos de código fuente. Rea-

lizan la compilación de los archivos, pero también el empaquetamiento en el formato adecuado para la entrega.

- **Herramientas de pruebas**. Herramientas para la definición de una estrategia de pruebas y para el seguimiento a lo largo del proyecto. Pueden ayudar en la definición de la prueba, la ejecución, la gestión de las pruebas y de los resultados obtenidos, etc.

- **Herramientas de desarrollo de interfaces gráficas de usuario**. Son herramientas que permiten desarrollar las interfaces gráficas de usuario de manera gráfica, de tal manera que, en lugar de la necesidad de programar el 100% de la interfaz, el desarrollador puede arrastrar componentes a las pantallas de manera visual y programar sólo el comportamiento dinámico.

- **Herramientas de medida de métricas**. Herramientas que permiten medir una serie de métricas de manera automatizada para darnos criterios objetivos, que a su vez nos permiten valorar la calidad del software desarrollado en cuanto a arquitectura, diseño, código, pruebas, etc.

- **Herramientas de gestión de la configuración y del cambio**. Herramientas que gestionan el cambio del software a lo largo de su vida. Pueden gestionar las distintas versiones que se van creando de cada artefacto, las distintas versiones del producto final, la integración de las nuevas versiones de cada parte del producto final, etc.

Para que el entorno de trabajo del ingeniero de software sea lo más adecuado posible, es necesario que las herramientas que éste usa se hallen correctamente integradas y se coordinen correctamente entre sí.

Entornos integrados de desarrollo

Como su nombre indica, los entornos integrados de desarrollo suelen integrar y aglutinar todo un conjunto de herramientas que resulten útiles para el diseño y la codificación de software. Suelen incorporar, por ejemplo, herramientas de depuración, de ejecución de pruebas, de medida de métricas y herramientas de desarrollo de la interfaz gráfica de usuario. Por otro lado, suelen coordinar el trabajo con las herramientas de gestión del cambio y de la configuración y permiten al usuario usar el control de versiones de manera integrada.

Otro caso típico de integración es el de las herramientas de modelización. Estas herramientas suelen incorporar herramientas de medida de métricas y herramientas de ingeniería inversa. Algunas, además, se integran con el entorno integrado de desarrollo, de tal manera que facilitan el paso de modelo a código o, incluso, mantienen los dos (código y modelos) actualizados, lo que permite que un cambio en el código se refleje en los modelos y viceversa.

Un último ejemplo: un entorno de integración continua es un entorno que integra una herramienta de control de las versiones, herramientas de pruebas y herramientas de construcción de software, para que, cada vez que se desarrolla una nueva versión de un archivo de código fuente, se elabore la entrega final y se ejecuten pruebas de manera automatizada.

5. Estándares de la ingeniería del software

La ingeniería del software, como disciplina de la ingeniería, requiere la existencia de estándares ampliamente aceptados que faciliten la comunicación y la interacción entre ingenieros.

Sin embargo, a veces los esfuerzos de estandarización se encuentran con problemas debido a la relativa juventud de nuestra disciplina y a la rapidez con la que avanza la tecnología y, por lo tanto, el dominio de aplicación de la ingeniería del software.

Por esta razón, la mayoría de los estándares en la ingeniería del software surgen rodeados de controversia, a pesar de que, a medida que avanza el tiempo, se va reconociendo su valor.

Uno de los organismos más activos en la definición de estándares para la ingeniería del software es el IEEE; más concretamente, la IEEE Computer Society. En el apéndice C de la guía SWEBOK (2004), podemos encontrar una lista con más de treinta estándares publicados por el IEEE relacionados con la ingeniería del software.

Otro organismo muy activo en esta área es el Object Management Group (OMG), que es un consorcio de empresas dedicado a la creación de estándares en tecnología orientada a objetos, como UML o CORBA.

A continuación, realizamos una compilación de los estándares que hemos considerado más importantes. Esta lista no pretende ser exhaustiva, sino simplemente indicativa de la variedad de estándares y organismos que están dedicados a la ingeniería del software.

5.1. Lenguaje unificado de modelización (UML)

El lenguaje unificado de modelización (UML) es un estándar publicado por el OMG que define la notación aceptada universalmente para la creación de modelos del software. Gracias a este estándar, cualquier ingeniero de software puede entender los modelos creados por otro ingeniero.

Ver también

Estudiaremos el lenguaje UML con más detalle en el módulo "Análisis UML".

La finalidad principal del lenguaje UML (y uno de los motivos de su éxito) es unificar la notación gráfica que usan los diferentes métodos de desarrollo con independencia del método empleado. De este modo, aunque un ingeniero no conozca el método de desarrollo con el que se ha llegado a crear un modelo, es perfectamente capaz de entender su significado.

Históricamente, UML se desarrolló dentro de la empresa Rational, creadora de la familia de métodos *unified process*, a pesar de que, actualmente, el OMG es quien controla totalmente la evolución del lenguaje.

Una característica interesante de UML es que (en parte debido a su independencia del método) no indica qué modelos o artefactos se pueden crear, sino que ofrece una serie de diagramas que los diferentes métodos pueden utilizar para sus artefactos. Así, un mismo tipo de diagrama se puede usar en diferentes contextos para generar artefactos diferentes.

5.2. *Software engineering body of knowledge* (SWEBOK)

El *software engineering body of knowledge* (SWEBOK) es un intento de establecer el conocimiento ampliamente aceptado por la comunidad de ingenieros del software.

Como parte de esta iniciativa, se publicó una guía que describe cuál es este cuerpo de conocimiento y dónde lo podemos encontrar.

Los objetivos de la guía del SWEBOK son:

- Promover una visión consistente de lo que es la ingeniería del software a escala mundial.

- Clarificar la situación (y las fronteras) de la ingeniería del software hacia otras disciplinas como las ciencias de la computación (*computer science*), la gestión de proyectos, la ingeniería de computadores (*computer engineering*) y las matemáticas.

- Caracterizar los contenidos de la disciplina de la ingeniería del software.

- Proporcionar un acceso organizado al cuerpo de conocimiento de la ingeniería del software.

La guía del SWEBOK organiza el conocimiento en diez áreas clave:

1) requisitos del software

2) diseño del software

3) construcción del software

4) pruebas del software

5) mantenimiento del software

6) gestión de la configuración del software

7) gestión de la ingeniería del software

8) proceso de la ingeniería del software

9) herramientas y métodos de la ingeniería del software

10) calidad del software

5.3. *Capability maturity model integration* (CMMI)

La *capability maturity model integration* (CMMI), publicada por el Software Engineering Institute, "se puede utilizar para guiar el proceso de mejora aplicado a un proyecto, una división o la organización completa". Por lo tanto, CMMI proporciona a los ingenieros del software una guía para la mejora de los procesos implicados en el desarrollo de software.

CMMI prevé, principalmente, tres áreas de actividad:

- **Gestión de servicios (CMMI-SVC)**, destinado a proveedores de servicios con el fin de ayudarlos a reducir costes, mejorar la calidad y la predictibilidad. Incluye buenas prácticas sobre qué servicios se deben ofrecer, asegurarse de que se está en condiciones de ofrecer un servicio y establecer acuerdos y actividades relacionadas, en general, con la provisión de servicios

- **Adquisición (CMM-ACQ)**, enfocado a ayudar a organizaciones que contratan proveedores de productos o servicios para mejorar las relaciones con los proveedores, los procesos de contratación y de control, así como la adecuación del producto o servicio adquirido a las necesidades de la organización.

- **Desarrollo (CMMI-DEV)**, dirigido a organizaciones que desarrollan software y que quieran mejorar la satisfacción de los clientes, la calidad del producto obtenido y el propio proceso de desarrollo.

Como parte del proceso de mejora CMMI define una serie de niveles de madurez, en función del nivel de implantación de las prácticas sugeridas.

5.4. *Project management body of knowledge* (PMBOK)

PMBOK (publicado por el Project Management Institute) es una guía de buenas prácticas para la gestión de proyectos. Como tal, quedaría fuera del ámbito de la ingeniería del software, tal y como la define la guía SWEBOK (2004), pero, según hemos indicado a lo largo del módulo, la mayoría de los desarrollos de software se llevan a cabo a modo de proyecto y, por lo tanto, la gestión del proyecto es clave para el éxito del desarrollo del software.

PMBOK identifica las áreas de conocimiento siguientes:

- gestión de la integración
- gestión del alcance
- gestión del tiempo
- gestión de la calidad
- gestión de los costes
- gestión del riesgo
- gestión de recursos humanos
- gestión de la comunicación
- gestión de las compras y adquisiciones

Resumen

Hemos visto que la ingeniería del software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, la operación y el mantenimiento de software. Este enfoque se adoptó con el fin de lograr los niveles de calidad, productividad y control de costes del resto de las ingenierías.

Sin embargo, el software, a diferencia de otros productos, es intangible y, por lo tanto, su producción no consume materias primas. Por otro lado, no se manufactura (todas las copias son idénticas y el coste de crear una es prácticamente nulo) ni se desgasta y, además, queda obsoleto rápidamente.

La aplicación de la ingeniería del software implica que debemos seguir un método, que describirá las características del proceso disciplinado que utilizaremos. Más concretamente, nos dirá qué tareas se deben llevar a cabo, quién las ha de realizar (los roles) y qué artefactos serán las entradas y los resultados de cada tarea.

De estos métodos de desarrollo, hemos visto que es importante escoger el más adecuado a la naturaleza del proyecto que se tenga que realizar, y hemos presentado tres grandes familias –los métodos que siguen el ciclo de vida en cascada, los métodos iterativos e incrementales y los métodos ágiles– y hemos visto un ejemplo representativo de cada una –Métrica 3, el proceso unificado y Scrum–.

Las tareas definidas por un método pueden pertenecer a ámbitos distintos, como la gestión del proyecto, la identificación y gestión de requisitos, la modelización, la construcción del software, las pruebas, la gestión de la calidad, el mantenimiento o la reingeniería.

En cuanto a los roles, cada método puede definir roles diferentes, a pesar de que hay ciertos roles que suelen aparecer con uno u otro nombre en la mayoría de los métodos. Es el caso del jefe de proyectos, el experto del dominio, el analista funcional, el arquitecto, el analista orgánico, el programador, el experto de calidad, el encargado de despliegue o el responsable del producto.

También hemos visto un resumen de algunas técnicas y herramientas propias de la ingeniería del software: la orientación a objetos, los bastimentos, el desarrollo basado en componentes, el desarrollo orientado a servicios, los patrones, las líneas de producto, la arquitectura dirigida por modelos y los lenguajes específicos del dominio, así como algunos estándares relacionados con las actividades mencionadas anteriormente.

Actividades

1. Poned dos ejemplos de cada uno de los tipos de software mencionados en el subapartado 1.3.

2. Dados los ejemplos siguientes de software, indicad, en cada caso, a qué tipo de software indicado en el subapartado 1.3 pertenece:

a) El núcleo del SO Linux
b) Un gestor de ventanas en un entorno de escritorio (por ejemplo, MetaCity)
c) Un paquete ofimático, como OpenOffice
d) El software de planificación de recursos empresariales SAP
e) Un sistema de gestión del correo electrónico vía web (como Hotmail)
f) La aplicación de campus virtual de la UOC
g) El software Mathematica
h) AutoCAD
i) El software que gestiona el GPS de un coche
j) El software de reconocimiento del habla

3. Dad dos argumentos a favor y dos argumentos en contra del hecho de tratar el desarrollo del software como una disciplina de la ingeniería.

4. Encontrad tres puntos en común y tres diferencias entre la ingeniería del software y el alpinismo.

5. ¿A qué tipo de actividad del subapartado 2.3 pertenecerían las tareas siguientes?

a) Estudiar el producto ofrecido por la compañía *X* para ver si satisface nuestras necesidades o si, por el contrario, deberemos buscar otro producto o hacer un desarrollo a medida.
b) Determinar si podemos acabar el proyecto en una fecha concreta si María deja el proyecto.
c) Establecer el calendario de reuniones de seguimiento del proyecto.
d) Estudiar cómo nos afecta la LOPD.
e) Crear un modelo de la interfaz de usuario para mostrar a los usuarios finales cómo será la aplicación.
f) Crear un diagrama UML con el modelo conceptual del dominio.
g) Determinar y documentar qué volumen máximo de usuarios debe soportar el sistema que hay que desarrollar.
h) Detectar cuáles son las necesidades de los administradores del sistema.
i) Elaborar un prototipo de la interfaz gráfica de usuario para probar diferentes combinaciones de colores y diferentes disposiciones de la información.
j) Diseñar un componente del sistema.
k) Comprobar que el sistema soporta el volumen máximo de usuarios que se determinó.
l) Verificar que se ha seguido el proceso de desarrollo tal y como está definido.
m) Recoger información sobre el número de requisitos implementados cada semana.
n) Corregir un error en un sistema ya en producción.

6. Clasificad, según la clasificación vista en el apartado 3.2, los proyectos siguientes:

a) Desarrollar una aplicación para sustituir la hoja de cálculo en la que apuntamos las vacaciones.
b) Desarrollar un sistema de comercio electrónico que permita incrementar las ventas en línea.
c) Evaluar OpenOffice como alternativa a las herramientas ofimáticas que se están usando actualmente en nuestra organización.
d) Identificar qué sistemas actuales nuestros podemos sustituir por aplicaciones de software libre.
e) Desarrollar una aplicación que incorpore datos de unos ficheros en un formato conocido y documentado y los guarde en una base de datos existente.
f) Desarrollar una aplicación para automatizar el proceso de compra de mercancías a proveedores en una organización en la que, actualmente, todo el proceso es manual.

7. Suponed que estamos trabajando en un pequeño proyecto con los requisitos (por orden de prioridad) y las valoraciones siguientes:

Requisito	Análisis y diseño	Implementación	Pruebas
a	1	1	1
b	3	5	2

Requisito	Análisis y diseño	Implementación	Pruebas
c	2	4	1
d	1	3	1
e	2	1	2
f	1	3	2
g	1	1	1

a) Planificad el proyecto suponiendo que la valoración se realiza en días de trabajo (por ejemplo, el requisito *a* necesita un día de análisis y diseño, un día de implementación y un día de pruebas) usando el ciclo de vida en cascada e indicando qué trabajo se hará cada semana.

b) Haced lo mismo con el ciclo de vida iterativo e incremental, suponiendo iteraciones de 10 días.

8. Indicad, para cada una de las situaciones siguientes, si creéis que cumplen o no los principios del Manifiesto ágil:

a) Una vez aclarada la funcionalidad en una conversación con el cliente, observamos que el coste de documentarla e implementarla es más o menos el doble de implementarla directamente y decidimos no documentarla.
b) El cliente debe firmar el documento de requisitos antes de empezar el desarrollo para asegurarnos de que lo ha entendido bien y que no habrá que introducir cambios.
c) Los desarrolladores deben rellenar un informe al final del día en el que indiquen, además de sus horas de trabajo, cuántas han dedicado a cada una de las tareas.
d) El jefe de proyectos no asigna tareas a los desarrolladores, sino que éstos se presentan voluntarios.
e) El cliente debe estar disponible en todo momento para resolver dudas a los desarrolladores (y no sólo al jefe de proyecto).
f) Se ha planificado en detalle y se ha documentado en un diagrama de Gantt todo el trabajo de los próximos seis meses.
g) Existe un proceso de gestión del cambio que permite introducir cambios en los requisitos una vez iniciado el desarrollo.
h) Todas las personas implicadas en el desarrollo del proyecto trabajan en la misma sala.
i) Toda la comunicación entre desarrolladores se debe llevar a cabo por medio de un foro para que quede registrada y se pueda reutilizar más adelante.
j) A la hora de medir el avance del proyecto no se tiene en cuenta la funcionalidad que está a medio implementar. Por ejemplo, si tenemos la especificación y el análisis de una funcionalidad pero no la hemos acabado de diseñar e implementar, consideramos que el avance es 0 y no el 50%. Por lo tanto, consideramos que el avance de una funcionalidad es SÍ/NO y no un porcentaje.

9. Métrica 3 es un método que se puede aplicar tanto al desarrollo estructurado como al desarrollo orientado a objetos. Indicad un artefacto que sea específico de cada uno de los casos indicados y otro que sea común a los dos casos.

10. Indicad cuáles son las tareas específicas del analista en OpenUP.

11. En el artículo "Metrics you can bet on", de Mike Cohn, el autor propone tres propiedades que deberían cumplir todas las métricas. ¿Cuáles son estas propiedades? Pensad en una métrica que hayáis usado (o que podríais usar en un proyecto) e indicad si cumple o no las propiedades mencionadas.

12. El método Scrum nos propone dos artefactos con nombre similar: el *product backlog* y el *sprint backlog*. Indicad cuál es la finalidad de cada uno y enumerad y justificad tres diferencias entre los dos artefactos.

13. Pensad otro ejemplo de separación entre interfaz e implementación en el mundo real.

14. Buscad un ejemplo de patrón de análisis. ¿Cómo está documentado el patrón? Pensad en un caso en el que pueda ser de aplicación.

15. Suponed que estamos desarrollando un producto y que nos piden un cambio en uno de los requisitos. Queremos determinar el impacto del cambio en nuestra planificación. ¿Cómo nos afecta en el supuesto de que estemos siguiendo el ciclo de vida en cascada? ¿Y en el

caso del ciclo de vida iterativo e incremental? ¿Cambia mucho el impacto según si ya está implementado o no?

16. Partiendo del ejemplo siguiente de programa escrito en un lenguaje específico del dominio, decid a qué dominio creéis que se aplica y qué ventajas e inconvenientes encontráis:

```
class
   NewTest def eBay_auto_feedback_generator
      open "http://my.ebay.com/ws/ebayisapi.dll?MyeBay"
      assertTitle "My eBay Summary"
      pause "2000"
      clickAndWait "link=Feedback"
      pause "3000"
      assertTitle "My eBay Account: feedback"
      clickAndWait "link=Leave Feedback"
      assertTitle "Feedback Forum: leave Feedback"
      click "//input[@name='which' and @value='positive']"
      type "comment", "Great eBayer AAAA++++"
      clickAndWait "//input[@value='Leave Feedback']"
      assertTitle "Feedback Forum: your feedback has been left"
   end
end
```

Ejercicios de autoevaluación

1. ¿Qué es el software para sistemas de información?

2. ¿Cuáles son las características inherentes al software que lo diferencian otros productos industriales?

3. ¿Por qué el desarrollo de software se organiza en forma de proyectos?

4. ¿Qué relación existe entre las tareas, los roles y los artefactos en un proceso de desarrollo?

5. Indicad cuatro tareas relacionadas con la actividad de gestión del proyecto.

6. ¿Cuál es la función de los requisitos en un proyecto de desarrollo?

7. ¿Cuál es la responsabilidad del analista funcional en un proyecto de desarrollo?

8. ¿Cuáles son las etapas del ciclo de vida en cascada?

9. ¿Cuáles son las principales ventajas del ciclo de vida iterativo e incremental respecto al ciclo de vida en cascada?

10. ¿Qué es y para qué sirve el plan de sistemas de información según Métrica 3?

11. ¿Cuáles son las prácticas fundamentales del proceso unificado?

12. ¿Cuáles son los artefactos de Scrum?

13. Indicad dos ventajas y un inconveniente de la reutilización de software.

14. ¿Cuál es la ocultación de información (en el contexto del software en general y de la orientación a objetos en particular)?

15. Indicad dos ventajas del uso de patrones.

16. ¿Qué son las herramientas CASE?

17. ¿Qué es UML?

Solucionario

Actividades

1. Software de sistemas: el núcleo de Linux, el controlador de la tarjeta de red.

Software de aplicación: OpenBravo (ERP), OpenOffice.

Software científico/de ingeniería: Mathematica, AutoCAD.

Software empotrado: el microsoftware de una cámara de fotos, el software que controla los frenos ABS de un coche.

Software de líneas de producto: según la web Software Product Line Conferences el software de los televisores Philips (que sería también un caso de software empotrado) deriva de una misma línea de productos. Otro ejemplo de la misma web es el software controlador de los productos RAID de la empresa LSI Logic: en este caso nos encontramos con un ejemplo de software de sistemas empotrado y de línea de producto.

Aplicaciones web: Facebook, Gmail.

Software de inteligencia artificial: ELIZA, un software desarrollado en los años sesenta que permitía mantener conversaciones con los usuarios. Por otro lado, la mayoría de los videojuegos de estrategia incluyen un componente de inteligencia artificial para hacer más realista el juego.

2.

a) Software de sistemas
b) Software de sistemas
c) Software de aplicación, línea de productos
d) Software de aplicación, línea de productos
e) Aplicación web
f) Aplicación web
g) Software científico/de ingeniería
h) Software de ingeniería
i) Software empotrado (también podría formar parte de una línea de productos)
j) Inteligencia artificial

3. Argumentos a favor:

- El principal argumento a favor es el gran número de proyectos de desarrollo de software que se han llevado a cabo a lo largo de los años aplicando la ingeniería al desarrollo del software y el modo como se ha mejorado respecto a la situación anterior. En el subapartado 1.5, hablamos de Chaos Report de los años 1995 y 2010 y de cómo se ha doblado en quince años el número de proyectos que se consideran 100% exitosos.
- Otro argumento a favor es que el enfoque sistemático (que, como acabamos de señalar, es viable) nos permite aplicar el conocimiento de las otras ingenierías al desarrollo del software. Es el caso, por ejemplo, de los principios del método de producción de Toyota, que se han trasladado al mundo del desarrollo de software y que se denomina *desarrollo lean*.

Argumentos en contra:

- Por un lado, Tom DeMarcos nos dice, en el artículo "Software Engineering: An Idea Whose Time Has Come and Gone?", publicado en la revista *IEEE Software*, en el número de julio/agosto del 2009, que si bien la ingeniería se centra en el control del proceso, éste no debería ser el aspecto más importante que considerar a la hora de pensar en el modo como desarrollamos el software, sino que nos deberíamos centrar en el modo como el software transforma el mundo en el que vivimos o la organización para la que lo desarrollamos.
- Por otro lado, otro argumento en contra del enfoque de ingeniería es aquel que considera que el impacto de las personas en el proceso de desarrollo de software (su habilidad, cómo se relacionan y el modo como se comunican, su estado de ánimo, su motivación, etc.) es tan grande que ningún método, por muy definido que esté, puede convertir el desarrollo de software en un proceso repetible.

4. Aspectos en común:

- Se suelen realizar en equipo.
- El nivel de experiencia y la habilidad natural de los participantes influyen mucho en el resultado final.

- Se debe conseguir un objetivo con un tiempo y unos recursos limitados.

Diferencias:

- En el alpinismo cada ascensión empieza de cero, mientras que la ingeniería del software puede aprovechar parte del trabajo que ya está hecho en otros proyectos y, por lo tanto, debemos intentar tener en cuenta los futuros desarrollos durante el proyecto.
- El alpinismo tiene un final claro y a corto plazo: o bien el momento de llegar a la cumbre o bien el momento de volver. La ingeniería del software, en cambio, debe tener en cuenta el mantenimiento futuro del sistema y su operación.
- En el alpinismo no es habitual cambiar de miembros del equipo durante la ascensión, mientras que, a lo largo de la vida del software, es habitual que se produzcan cambios.

5.

a) Gestión del proyecto
b) Gestión del proyecto
c) Gestión del proyecto
d) Gestión del proyecto, requisitos
e) Modelización
f) Modelización
g) Requisitos
h) Requisitos
i) Requisitos
j) Construcción y pruebas
k) Construcción y pruebas
l) Calidad
m) Calidad
n) Mantenimiento

6.

a) 1 (Objetivo claro y solución conocida)
b) 2 (Objetivo claro y solución no conocida)
c) 1 (Objetivo claro y solución conocida)
d) 3 (Objetivo poco claro y solución no conocida)
e) 1 (Objetivo claro y solución conocida)
f) 2 (Objetivo claro y solución no conocida)

7. a) Cada semana son 5 unidades de valoración.

Semana 1: análisis y diseño de a, *b* y parte de *c* (1/2).
Semana 2: análisis y diseño de parte de *c* (1/2), *d*, *e* y *f*.
Semana 3: análisis y diseño de *g*. Implementación de *a* y parte de *b* (3/5).
Semana 4: implementación de parte de *b* (2/5) y parte de *c* (3/4).
Semana 5: implementación de parte de *c* (1/4), *d* y *e*.
Semana 6: implementación de *f* y *g*. Pruebas de *a*.
Semana 7: pruebas de *b*, *c*, *d* y parte de *e* (1/2).
Semana 8: pruebas de e (1/2), *f* y *g*.

b) En este caso no distinguimos análisis y diseño, implementación y pruebas, sino que planificamos por requisitos analizados, implementados y probados.

Iteración 1 (semanas 1 y 2): requisito *a, c* (el *b* no lo podemos completar en esta iteración; por ello, hacemos el *c* antes).
Iteración 2 (semanas 3 y 4): requisito *b*.
Iteración 3 (semanas 5 y 6): requisitos *d* y *e*.
Iteración 4 (semanas 7 y 8): requisitos *f* y *g*.

8.

a) No sigue los principios del manifiesto ágil porque este, aunque dice que preferimos software funcionando sobre documentación exhaustiva, también dice que reconoce el valor de los ítems a la derecha y, por lo tanto, no hacer ninguna documentación no va en contra de esta última frase.
b) No cumple el principio "Colaboración con el cliente por encima de negociación del contrato".
c) En este caso, dependerá de cuánto tiempo deben dedicar los desarrolladores a rellenar el informe, dado que, en función de esto, se puede considerar que no se cumple el principio "Individuos e interacciones por encima de procesos y herramientas".
d) Sí que cumple el principio "Individuos e interacciones por encima de procesos y herramientas".

e) Sí que cumple el principio de "Colaboración con el cliente por encima de negociación del contrato".
f) No cumple el principio "Responder al cambio por encima del seguimiento de un plan".
g) Sí que cumple el principio "Responder al cambio por encima del seguimiento de un plan".
h) Sí que cumple el principio "Individuos e interacciones antes de procesos y herramientas".
i) No cumple el principio "Individuos e interacciones antes de procesos y herramientas".
j) Sí que cumple el principio "Software que funciona antes de la documentación exhaustiva".

9. En la versión estructurada, hay un artefacto denominado *Modelo lógico de datos normalizado* que no existe en la versión OO. En cambio, la versión OO tiene un artefacto conocido como *Modelo de clases de análisis* que no existe en la versión estructurada. En cambio, el "Catálogo de requisitos" es común a los dos enfoques.

10. En OpenUP, el analista es responsable de lo siguiente:

- identificar y dar una visión general de los requisitos
- detallar los requisitos globales del sistema
- detallar los casos de uso
- desarrollar la visión tecnológica

11. Las propiedades que define el artículo son:

- Las métricas deben medir cosas medibles.
- Las métricas se deben tomar en el nivel adecuado y en la unidad adecuada.
- Sólo tiene sentido medir si se quiere actuar sobre el resultado.

Un ejemplo típico de métrica es el número de líneas de código de un programa. Son fáciles de medir (cumplen el primer principio) y es fácil de determinar la unidad (líneas de código o miles de líneas de código, según el tamaño del programa).

Sin embargo, se debe tener en cuenta que los programas más complejos necesitarán, a priori, más líneas de código. Por ello, si queremos seguir el segundo principio, a la hora de medir variaciones en el número de líneas de código debemos tener en cuenta la variación en funcionalidad del programa.

Finalmente, en cuanto al tercer principio, sabemos que un incremento en el número de líneas de código que no sea proporcional al incremento en la funcionalidad del programa suele estar asociado a un exceso de repetición dentro del código y a un descenso en la calidad de éste, por lo que, si se detecta esta situación, habrá que actuar para reducir el volumen de código que hay que mantener.

12. Ambos son listas de cosas que se deben hacer antes de conseguir un objetivo concreto (sacar una nueva versión del producto en el caso del *product backlog* o finalizar con éxito una iteración en el caso del *spring backlog*); sin embargo, mientras que en el *product backlog* lo que tenemos son funcionalidades, errores que corregir y mejoras, en el *sprint backlog* tenemos tareas concretas (es decir, el *sprint backlog* contiene la descomposición de los ítems del *product backlog*).

13. Otro ejemplo de separación entre interfaz e implementación podría ser una calculadora: la interfaz sería el conjunto de teclas con las que introducimos los números y los signos de las operaciones y la implementación sería el conjunto de circuitos que forman la calculadora, así como el software que la controla.

14. En la web de Martin Fowler podemos encontrar el patrón *rango*, que sirve para representar rangos de datos, como el periodo de validez de una oferta en una aplicación de comercio electrónico.

15. En el caso del ciclo de vida en cascada, si ya hemos superado la etapa de requisitos, habrá que modificar el documento de requisitos del sistema y propagar el cambio en los documentos de análisis, diseño, implementación y pruebas que corresponda, según en qué etapa del proceso nos encontremos.

En el caso del ciclo de vida iterativo e incremental, dependerá fundamentalmente de si hemos implementado o no el requisito. Si ya lo hemos implementado, el coste será similar al caso anterior (volver a hacer requisitos, análisis, diseño, implementación y pruebas), mientras que si no lo hemos implementado todavía, sólo habrá que revisar la valoración, dado que no necesitaremos detallarlo hasta llegar a la iteración en la que se deba implementar.

16. Se trata de un programa escrito en un lenguaje de pruebas de aplicaciones web. La principal ventaja de usar un lenguaje específico del dominio, en este caso, es que es bastante

fácil hacerse una idea de cuál es el comportamiento que está simulando el programa al leer el código fuente.

Ejercicios de autoevaluación

1. El software para sistemas de información es un tipo de software que gestiona una cierta información mediante un sistema gestor de bases de datos y da apoyo a una serie de actividades humanas dentro del contexto de un sistema de información. (Podéis ver el subapartado 1.3).

2. El software es intangible, no se manufactura, no se desgasta y, además, queda obsoleto rápidamente. (Podéis ver el subapartado 1.6).

3. El desarrollo tiene un inicio y un final claros y, además, proporciona un resultado único. Por ello no se organiza de manera continua, sino como proyectos. (Podéis ver el subapartado 2.1).

4. Las personas, en función de su **rol**, deben llevar a cabo una serie de **tareas** que tendrán como punto de partida un conjunto de **artefactos** y generarán, como resultado, otro conjunto de **artefactos**. (Podéis ver el subapartado 2.2).

5. Estudio de viabilidad, valoración, definición de los objetivos, formación del equipo de trabajo, establecimiento de hitos e identificación de riesgos. (Podéis ver el subapartado 2.3.1).

6. Los requisitos expresan las necesidades y restricciones que afectan a un producto de software que contribuye a la solución de un problema del mundo real. (Podéis ver el subapartado 2.3.2).

7. El analista funcional es el responsable de unificar las diferentes visiones del dominio en un único modelo que sea claro, conciso y consistente. (Podéis ver el subapartado 2.4.3).

8. Las etapas del ciclo de vida en cascada son: requisitos, análisis y diseño, implementación, pruebas y mantenimiento. (Podéis ver el subapartado 3.2.1).

9. El ciclo de vida iterativo e incremental acelera la retroalimentación, dado que cada iteración cubre todas las actividades del desarrollo. Además, en todo momento se tiene un producto operativo, con lo que se puede acelerar el regreso de la inversión. (Podéis ver el subapartado 3.2.2).

10. El plan de sistemas de información tiene como finalidad asegurar que el desarrollo de los sistemas de información se lleva a cabo de manera coherente con la estrategia corporativa de la organización. El plan de sistemas de información, por lo tanto, es la guía principal que deben seguir todos los proyectos de desarrollo que empiece la organización. Como tal, incorpora un catálogo de requisitos que han de cumplir todos los sistemas de información, una arquitectura tecnológica, un modelo de sistemas de información, etc. (Podéis ver el subapartado 3.3.1).

11. Las seis prácticas fundamentales del proceso unificado son: desarrollo iterativo e incremental, gestión de los requisitos, arquitecturas basadas en componentes, utilización de modelos visuales, verificación de la calidad del software y control de los cambios al software. (Podéis ver el subapartado 3.3.2).

12. Scrum define tres artefactos: el *product backlog*, el *sprint backlog* y el *burn down chart*. (Podéis ver el subapartado 3.3.3).

13. Como ventajas tenemos la oportunidad, la disminución de costes, la fiabilidad y la eficiencia. Uno de los problemas sería que hay que hacer una gestión exhaustiva de las diferentes versiones del componente. (Podéis ver el subapartado 4.1).

14. La ocultación de información consiste en esconder los detalles sobre la estructura interna de un módulo, de manera que podemos definir claramente qué aspectos de los objetos son visibles públicamente (y, por lo tanto, utilizables por los reutilizadores) y cuáles no. (Podéis ver el subapartado 4.1.1).

15. Las principales ventajas del uso de patrones son: reutilizar las soluciones, beneficiarnos del conocimiento previo, comunicar y transmitir nuestra experiencia, establecer un vocabulario común y encapsular conocimiento detallado sin la necesidad de reinventar una solución al problema. (Podéis ver el apartado 4.1.5).

16. Las herramientas CASE son aplicaciones informáticas que ayudan al ingeniero del software a aplicar los principios de la ingeniería al desarrollo de software. (Podéis ver el subapartado 4.3).

17. El lenguaje unificado de modelización (UML) es un estándar publicado por el OMG que define la notación aceptada universalmente para la creación de modelos del software. (Podéis ver el subapartado 5.1).

Glosario

abstracción *f* La abstracción consiste en identificar los aspectos relevantes de un problema, de manera que nos podamos concentrar sólo en ellos.

actividad *f* Conjunto de tareas que llevan a cabo las personas que desempeñan un determinado rol.

análisis *m* El análisis de un sistema informático documenta cómo debe ser el producto que hay que desarrollar desde un punto de vista externo (es decir, sin considerar cómo está hecho por dentro). Normalmente, esta documentación se basa en modelos.

artefacto *m* Objeto producido por el trabajo del ser humano. En el contexto de la ingeniería del software, cada uno de los documentos, modelos, programas, etc., que se generan como resultado del trabajo del ingeniero.

ciclo de vida *m* El ciclo de vida de un proceso define cuáles son las diferentes etapas por las que va pasando un proceso a lo largo de su vida. En la ingeniería del software usamos este término para clasificar los métodos en familias según el tipo de ciclo de vida.
Ved **ciclo de vida en cascada**, **ciclo de vida iterativo** y **ciclo de vida incremental**

ciclo de vida en cascada *m* Ciclo de vida secuencial que consta de las etapas siguientes: requisitos, análisis y diseño, implementación, pruebas y mantenimiento. A pesar de haber varias variantes del ciclo de vida en cascada, lo que las caracteriza a todas es su naturaleza secuencial.

ciclo de vida incremental *m* Un ciclo de vida es incremental cuando el producto que hay que desarrollar se crea intermediando pequeños incrementos de funcionalidad completa que se van agregando al producto desarrollado.

ciclo de vida iterativo *m* Un ciclo de vida es iterativo si organiza el desarrollo en iteraciones de tiempo determinado. La principal ventaja del ciclo de vida iterativo es que permite establecer puntos de control regulares (al final de cada iteración) durante el desarrollo.

computer aided software engineering Término que se usa normalmente para referirse a las herramientas cuya finalidad es ayudar el ingeniero de software a aplicar los principios de ingeniería en el desarrollo de software.
Sigla **CASE**

construcción del software *f* Actividad cuya finalidad es la obtención del producto ejecutable. Esto incluye, entre otras funciones, la creación del código fuente, la creación de los archivos ejecutables o la gestión de la configuración.

desarrollo ágil *m* El desarrollo ágil consiste en aplicar los cuatro principios del Manifiesto ágil al desarrollo de software. También se conoce como desarrollo ágil el conjunto de métodos de desarrollo que comparten los principios mencionados.

desarrollo *lean* *m* El desarrollo *lean* intenta aplicar las técnicas de manufactura *lean* al desarrollo de software. Estas técnicas se derivan, originariamente, del sistema de producción Toyota (TPS).

desarrollo de software *m* Acto de producir o crear software.

diseño de software *m* El diseño de software documenta la estructura y el comportamiento interno de un sistema informático.

gestión del proyecto *f* Todo aquello que no sea la ejecución del proyecto. Su objetivo principal es asegurar el éxito del proyecto.

hardware *m* Conjunto de instrucciones codificadas y de datos que son la expresión completa de un procedimiento, y en particular de un algoritmo, ejecutable por un sistema informático.

implementación (como actividad del desarrollo) *f* Creación del código fuente.

implementación (como parte de la ocultación de información) *f* Parte interna de un objeto o un sistema; incluye todo aquello que no es visible a quienes lo usan.

ingeniería *f* Aplicación de un enfoque sistemático, disciplinado y cuantificable a las estructuras, las máquinas, los productos, los sistemas o los procesos para obtener un resultado esperado.

ingeniería del software *f* Aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, la operación y el mantenimiento del software; es decir, la aplicación de la ingeniería al software.

interfaz (como parte de la ocultación de información) *f* Parte externa de un objeto o un sistema; incluye todo aquello sí que es visible a quienes lo usan.

mantenimiento del software *m* Comprende la modificación posterior al desarrollo del producto de software para corregir los errores o adaptarlo a nuevas necesidades.

método *m* Definición de las características del proceso o procedimiento disciplinado utilizado en la ingeniería de un producto o en la prestación de un servicio.

metodología *f* Ciencia que estudia los métodos. Conjunto de los métodos de una disciplina.

modelización *f* Actividad que consiste en la creación de modelos del sistema que hay que desarrollar: estos modelos facilitarán la comprensión de los requisitos y el diseño del sistema.

ocultación de información *f* La ocultación de información consiste en esconder los detalles sobre la estructura interna de un módulo, de manera que podamos definir claramente qué aspectos de los objetos son visibles públicamente (y, por lo tanto, utilizables por los reutilizadores) y cuáles no.

operación del software *f* La operación del software consiste en ejecutar el producto de software dentro de su entorno en ejecución para llevar a cabo su función.

procedimiento *m* Ved **proceso**

proceso *m* Manera de descomponer una acción progresiva.

programación *f* Normalmente, se denomina programación a la creación del código fuente.

requisito *m* Los requisitos expresan las necesidades y restricciones que afectan a un producto de software que contribuye a la solución de un problema del mundo real.

sistema de información *m* Cualquier combinación de tecnología de la información y actividades humanas que utilizan esta tecnología para dar apoyo a la operación, gestión o toma de decisiones.

software *m* Conjunto de programas. Más formalmente, conjunto de programas de computación, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de cómputo.

tarea *f* Como parte de la ejecución de un proceso, cada tarea debe ser llevada a cabo por una persona con un rol concreto para crear unos artefactos de salida a partir de unos artefactos de entrada.

unified modeling language *m* Lenguaje unificado de modelización. Es un estándar publicado por el OMG que define la notación aceptada universalmente para la creación de modelos del software.
Sigla **UML**

Bibliografía

Bibliografía principal

Pressman, R. S. (2005). *Ingeniería del Software* (6.ª ed.). McGraw Hill.

Este libro es una buena introducción al estudio de la ingeniería del software en general.

VV. AA. (2004). *SWEBOK. Software Engineering Body Of Knowledge Guide*. IEEE Computer Society.

Esta obra contiene el conocimiento fundamental (es decir, aquellas ideas que son ampliamente aceptadas en la industria) de la ingeniería del software. Se puede consultar en la dirección:

http://www.computer.org/portal/web/swebok/htmlformat (Última visita: septiembre del 2010).

Bibliografía complementaria

Wysocki, R. K. (2009). *Effective Project Management*: *Traditional, Agile, Extrem* (5.ª ed.). Wiley.

Para el estudio de los diferentes métodos de desarrollo y para saber cuándo es más conveniente aplicar uno u otro.

Cockburn, A. (2007). *Agile Software Development*: *The Cooperative Game* (2.ª ed.). Addison-Wesley.

Para reflexionar sobre la naturaleza del desarrollo del software, el estudio de varias metáforas y el desarrollo ágil de software.

Página oficial del método Métrica 3. http://www.csae.map.es/csi/metrica3/index.html (Última visita: septiembre del 2010).

De esta página podéis bajar los documentos que describen el método. Es especialmente interesante el documento de introducción porque nos da una visión general del método y nos ayuda a situar en contexto el resto de los documentos.

Wiki del método OpenUP. http://epf.eclipse.org/wikis/openup/ (Última visita: septiembre del 2010).

Wiki del método OpenUP, variante ágil del proceso unificado. Este método está publicado bajo licencia EPL y, por lo tanto, es públicamente accesible.

Scrum Guide. http://www.scrumguides.org/scrum-guide.html (Última visita: septiembre del 2010).

Este breve documento escrito por uno de los creadores de Scrum nos ofrece una visión más detallada del método ágil de desarrollo Scrum.

Página oficial del OMG sobre MDA. http://www.omg.org/mda/ (Última visita: septiembre del 2010).

Página oficial del OMG sobre MDA en la que podemos encontrar documentación relacionada con el estándar y también la versión oficial.

Referencias bibliográficas

Fowler, M. "Language Workbenches: The Killer-App for Domain Specific Languages?". http://martinfowler.com/articles/languageworkbench.html (Última visita: septiembre del 2010).

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1994). *Design Patterns: elements of Reusable Object-Oriented Software*. Addison-Wesley.

Meyer, B. (1999). *Construcción de Software Orientado a Objetos*. Prentice Hall.

Poppendieck, M.; Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*. Addison-Wesley.

Rational Unified Process: BEST Practices for Software Development Teams

http://www.ibm.com/
developerworks/rational/library/content/03july/1000/1251/1251_bestpractices_TP026B.pdf
(Última visita: septiembre del 2010).

Toyota Production System. http://www2.toyota.co.jp/en/vision/production_system/
index.html (Última visita: septiembre del 2010).