por Nacho Iborra

Entornos de Desarrollo
Bloque 1
Tema 3: Etapas del desarrollo software. Metodologías

1.3.1. Ingeniería del software

Para desarrollar software correctamente es recomendable seguir una serie de pasos, aplicar un enfoque concreto para dicho desarrollo. La rama de la informática que nos ayuda a aplicar correctamente estos enfoques y seguir los pasos adecuados se denomina **ingeniería del software**.

1.3.1.1. Etapas en el desarrollo software

En (casi) cualquier proceso de ingeniería del software para desarrollar un producto concreto, se llevan a cabo las siguientes etapas:

1. **Análisis de requisitos**: esta etapa incluye la comunicación con el cliente para establecer qué aplicación quiere, y una primera fase de análisis para esbozar el comportamiento de la misma. Puede subdividirse en dos etapas:
    - **Especificación de requisitos**: la comunicación con el cliente propiamente dicha. En esta etapa se establecerán entrevistas u otros métodos de obtención de información del cliente, procurando dejar claros todos los aspectos o necesidades que debe cumplir el software a desarrollar. Una vez obtenidos los requisitos, se elabora un documento lo más completo posible, denominado normalmente *especificación de requisitos*.
    - **Análisis**: a partir de la especificación de requisitos anterior, se elabora un documento donde, ayudándonos de algunos diagramas visuales, detallamos qué debe cumplir la aplicación a desarrollar (qué funcionalidades debe proporcionar, a quién, qué operaciones dependen de qué otras, etc.).
2. **Diseño**: a partir del análisis del paso anterior, se determina cómo funcionará el software de forma general. Aquí se elaborarán otra serie de diagramas que luego permitirán implementar el programa de acuerdo a ellos, tales como los diagramas de clases, secuencia, etc.
3. **Programación o implementación**: a partir del diseño previo y los diagramas obtenidos, se pasará a implementar la aplicación, utilizando para ello un lenguaje (o lenguajes) de programación determinado(s).

4. **Pruebas**: una vez implementado el producto, se debe probar su funcionamiento, comprobando que realiza las tareas correctamente, y que no hay ningún caso en que pueda fallar. Se considera una buena práctica que estas pruebas las realice alguien distinto a quien programó la aplicación.
5. **Mantenimiento**: esta etapa consiste en mejorar el software desarrollado, añadiendo ampliaciones debidas a nuevos requisitos, o corrigiendo posibles errores detectados en la fase de pruebas.

Resumiendo estas etapas, podríamos decir que el enfoque que proporciona la ingeniería del software nos ayuda a entender el problema a resolver (análisis de requisitos), diseñar la posible solución, implementarla, probarla y después gestionar las tareas previas para conseguir una mayor calidad (mantenimiento).

Sin embargo, todas estas etapas se ven a menudo como un freno. Muchos desarrolladores perciben la ingeniería del software como algo demasiado formal y estructurado, y eso les impide desarrollar los productos rápidamente, que es lo que exigen cada vez más clientes. Debemos contemplar la ingeniería del software como algo adaptativo, ya que proporciona diferentes modelos y metodologías para adaptarlas al ritmo de desarrollo que necesitamos, como veremos más adelante.

1.3.2. Ciclos de vida del software

Un ciclo de vida es un conjunto de fases por las que debe pasar un sistema (en este caso un proyecto software), desde que nace hasta que finaliza su uso. En cada ciclo de vida se establecen tanto las fases por las que debe pasar, como los criterios para pasar de una fase a la siguiente, incluyendo qué entradas y salidas produce cada fase, y las actividades a realizar en cada fase.

A los productos que se generan después de cada fase (materiales o inmateriales) se les suele denominar *entregables* y, o bien forman parte de la entrada de la siguiente fase, o bien permiten evaluar el estado del proyecto en un punto determinado.

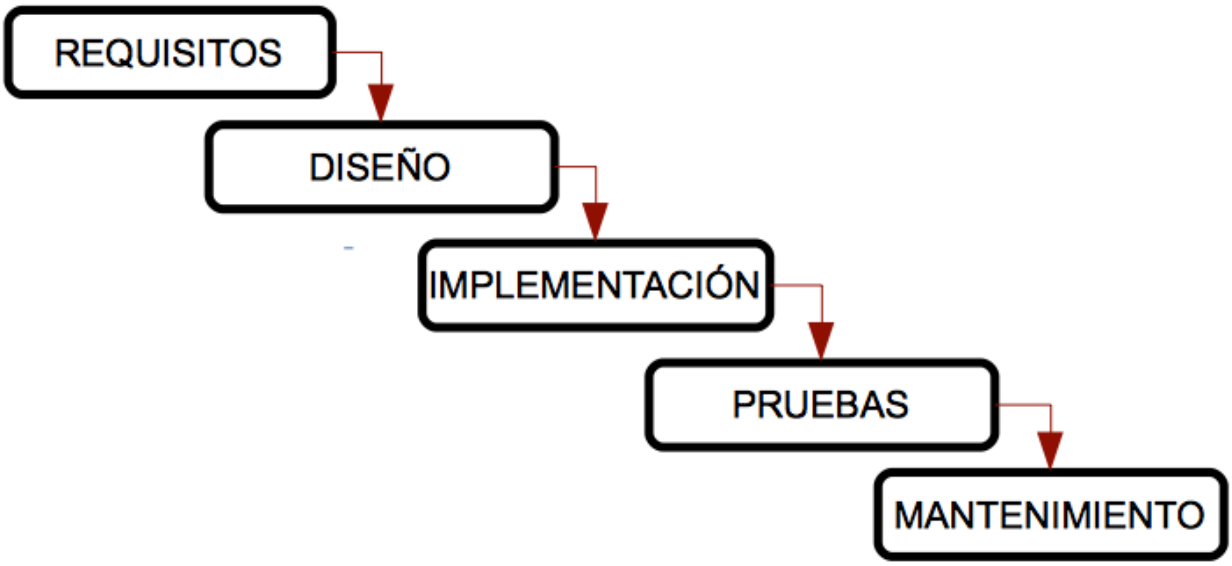
Algunos ciclos de vida son repetitivos, es decir, podemos pasar por una misma fase varias veces, en función del estado en que se encuentra el producto en cada momento, en un proceso que se conoce como *realimentación*.

Veremos a continuación algunos de los modelos de ciclos de vida más habituales en el desarrollo de productos software, comentando sus ventajas e inconvenientes. En todos ellos entran en juego las etapas del desarrollo software vistas antes (análisis de requisitos, diseño, programación, etc.), o bien alguna variante de las mismas.

1.3.2.1. Modelo en cascada

Este modelo es el más antiguo y ampliamente difundido de los que existen. Ideado por W. Royce en los años 70, ordena las etapas del software rigurosamente, de forma que el inicio

de una etapa debe esperar a la finalización de la anterior.



Se denomina modelo en cascada porque las etapas se colocan una debajo de la otra, y el desarrollo va fluyendo hacia abajo por las distintas fases, como si fuera una cascada.

Ventajas:

- Apropiado para proyectos pequeños y estables, donde los requisitos se definen perfectamente al inicio
- Modelo bien organizado, donde no se mezclan fases
- Simple y fácil de usar, gracias en parte a su rigidez

Inconvenientes:

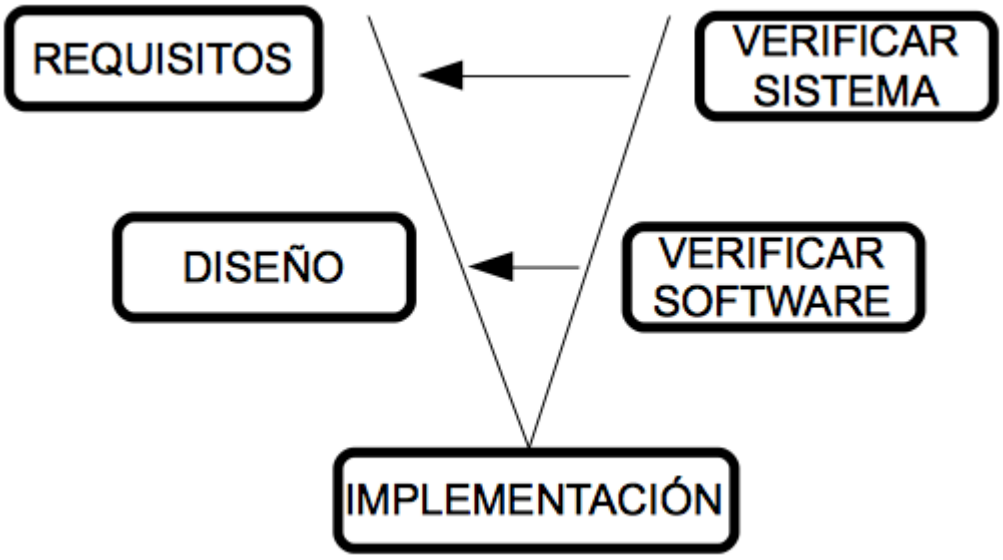
- Poco aplicable a la mayoría de proyectos en la vida real (casi ningún cliente deja bien especificados los requisitos al principio)
- Los resultados no son visibles hasta el final, lo que puede generar cierta inquietud en el cliente
- Es poco viable pretender tener una fase perfecta antes de pasar a la siguiente
- Los fallos no se detectan hasta la fase de pruebas, en el tramo final del proyecto

Variantes:

Existen algunas variantes sobre este modelo en cascada, como por ejemplo el **modelo Sashimi**, llamado así porque las fases se superponen temporalmente, como el pescado japonés. En este modelo existe cierto solape temporal entre una fase y la siguiente, de forma que, mientras se están definiendo los requisitos, se comienza con el diseño (pudiendo así ampliar o modificar los requisitos a medida que se avanza en el diseño). De la misma forma, mientras se está diseñando, se comienza a implementar (pudiendo haber cambios en el diseño motivados por cuestiones de implementación), etc.

1.3.2.2. Modelo en V

Uno de los problemas del modelo tradicional en cascada es que los fallos en el software no se detectan hasta las fases finales. Con el modelo en V, las pruebas se inician lo más pronto posible y, además, estas pruebas deben hacerse en paralelo por otro equipo. Así, las pruebas se integran en cada fase del ciclo de vida.



La parte izquierda de la V representa el análisis de requisitos, diseño e implementación del sistema, y la parte derecha representa la integración de las partes del sistema y su verificación o prueba. Se va avanzando por la izquierda hasta llegar a la punta inferior (la codificación), y después se llevan a cabo las validaciones o pruebas de la derecha. En cuanto se detecte algún problema, se vuelve al lado izquierdo en el punto donde se encontró.

Ventajas:

- Simple y fácil de usar, como el modelo en cascada
- En cada fase hay entregables específicos a realizar
- Mayor probabilidad de éxito que el modelo en cascada, gracias a los planes de prueba asociados a cada etapa del proceso
- Útil para proyectos pequeños con requisitos fáciles de entender

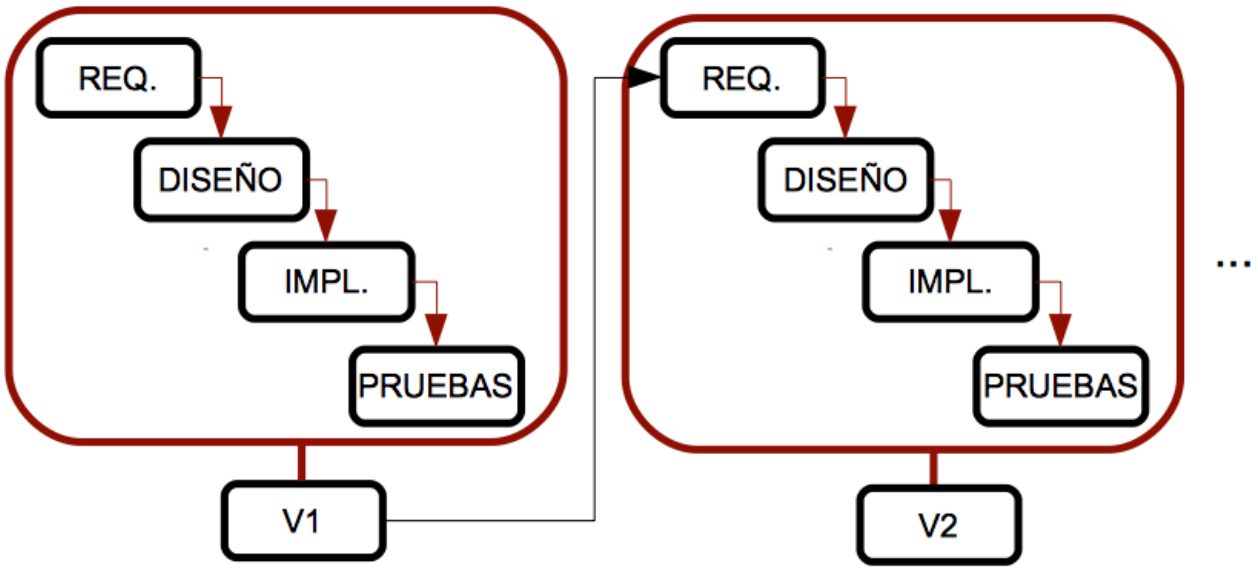
Inconvenientes:

- Rígido, como el modelo en cascada
- El usuario sigue sin ver resultados hasta fases tardías, ya que no se desarrolla ningún prototipo intermedio
- No hay a veces caminos claros para pasar del lado derecho de la V al izquierdo

1.3.2.3. Modelo iterativo

Uno de los principales problemas de los modelos anteriores es que sólo funcionan bien cuando los requisitos son sencillos o están bien especificados, pero esa no suele ser la tónica habitual en los proyectos software.

Para intentar paliar esto, el modelo iterativo propone una especie de repetición del modelo en cascada, generando tras cada repetición una versión o prototipo del proyecto. Esta versón se revisa, se detectan los fallos y se vuelve sobre los requisitos para pulirlos.



Ventajas:

- No hace falta definir los requisitos totalmente al principio
- Se gestionan mejor los riesgos, al hacer pequeñas entregas parciales intermedias

Inconvenientes:

- El no necesitar tener los requisitos definidos al principio puede acarrear que luego surjan requisitos no contemplados que afecten significativamente al diseño o arquitectura general de la aplicación

Variantes:

Existen algunas variantes de este modelo iterativo, con otros nombres y algunas peculiaridades que las diferencian de éste:

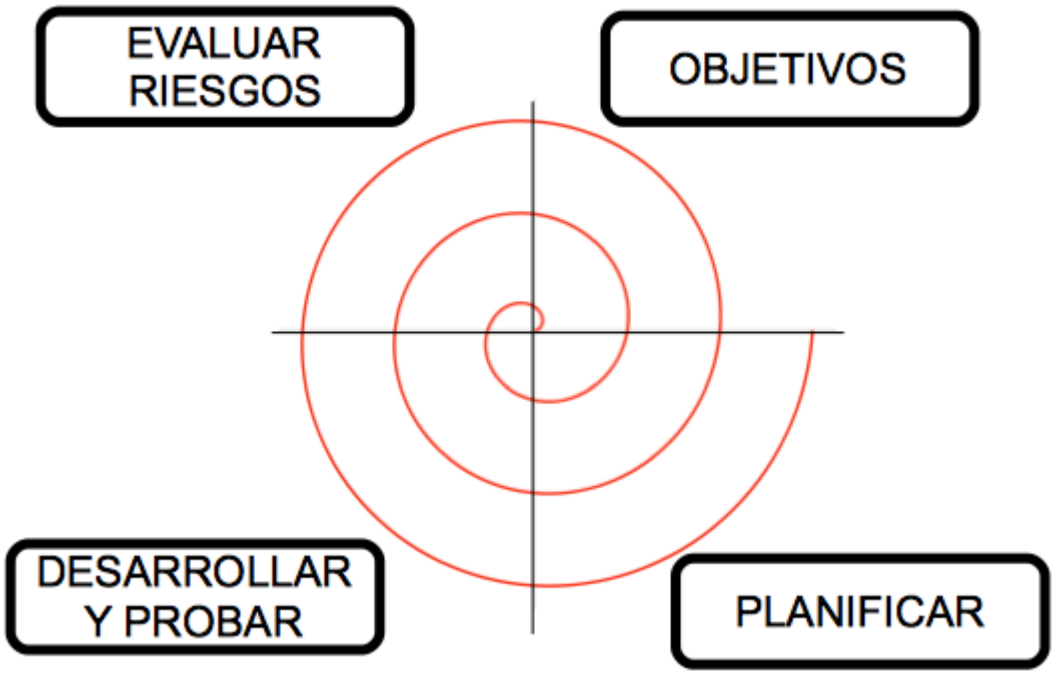
- **Modelo incremental**: cada versión que se entrega es un producto con pequeñas mejoras o cambios respecto al anterior. A diferencia del modelo iterativo, las versiones que se entregan son operativas, pero con funcionalidades muy limitadas al principio, y con pequeños cambios entre versiones. Esto lo hace más fácil de probar que el modelo iterativo (hay pocos cambios que verificar de una versión a la siguiente). Sin embargo, como principal inconveniente se tiene que se requiere una gran experiencia para saber elaborar los prototipos de cada entrega de forma proporcionada.

- **Modelo basado en prototipos**: se desarrollan prototipos de la aplicación, al principio con una recolección rápida de requisitos y un diseño simple, para que el cliente los vaya supervisando desde fases tempranas, y pueda aportar su opinión o nuevos requisitos no contemplados. Como ventaja principal se tiene la participación del cliente desde fases tempranas en la evaluación de prototipos. Como inconvenientes, puede ser un modelo costoso, donde se invierta en prototipos que luego se desechen. Además, el cliente puede frustrarse por ver versiones de la aplicación que no terminan de funcionar como él quiere, y el desarrollador puede tener la tentación de dar un "empujón" a la aplicación para añadir todo lo que el cliente quiere, saltándose las pautas de calidad y mantenimiento establecidas.

1.3.2.4. Modelo en espiral

Propuesto por Boehm en 1988, en el modelo en espiral las fases o actividades a realizar se disponen en una espiral dividida en cuadrantes, de forma que en cada cuadrante se realiza una actividad, y cada vuelta de la espiral pasa por todas las actividades.

Este modelo, al igual que el anterior, tiene en cuenta el riesgo que aparece al desarrollar software. Así, comenzando desde el centro de la espiral, en cada vuelta de la misma se analizan las posibles alternativas de desarrollo, se opta por los riesgos más asumibles y se hace un ciclo de dicha espiral. Si el cliente propone mejoras o correcciones sobre el producto que se obtiene, se vuelven a evaluar alternativas y riesgos y se realiza otra vuelta, hasta que el producto sea aceptado.



- En la fase de **objetivos** se establecen los productos a obtener (requisitos, especificación, etc.)

- En la fase de **evaluar riesgos** se identifican los posibles riesgos del proyecto, y se seleccionan una o varias alternativas para reducirlos o eliminarlos
- En la fase de **desarrollar y probar** se llevan a cabo las etapas de diseño, implementación y prueba, según las alternativas elegidas previamente en el análisis de riesgos
- En la fase de **planificar** se revisa lo realizado, se coteja con el cliente y se decide si hace falta depurar o ampliar alguna opción de la aplicación, y en consecuencia dar una vuelta más a la espiral.

Ventajas:

- Adecuado para proyectos largos y complicados
- Se reducen los riesgos del proyecto
- Integra el desarrollo con el mantenimiento del producto
- Se producen prototipos o versiones en etapas tempranas, de forma que los pueda ver el cliente y aportar información durante el desarrollo

Inconvenientes:

- Se requiere experiencia en el equipo de desarrollo para evaluar correctamente los posibles riesgos
- Se genera bastante trabajo adicional
- Puede resultar bastante costoso
- No es apropiado para proyectos pequeños

> Ejercicios propuestos:

> 1.3.2.1. Realiza una tabla como la siguiente (en papel, o en un editor de textos) y rellena cada casilla con un SI o un NO, dependiendo de si la característica citada en la primera columna corresponde al modelo de ciclo de vida indicado en la primera fila.

	Cascada	**V**	**Incremental**	**Espiral**
Es simple				
Genera versiones intermedias del producto				
Válido para proyectos de requisitos inciertos				
Permite valorar bien los riesgos de la solución elegida				

> 1.3.2.2. Nuestra empresa va a desarrollar una aplicación para llevar la contabilidad de una cadena de cines. El cliente aún no tiene demasiado claro qué quiere que haga el sistema, y se quiere utilizar una tecnología novedosa con la que aún no tenemos demasiada experiencia. A juzgar por lo obtenido en la tabla del ejercicio anterior, valora qué modelo(s) de ciclo de vida emplearías y cuáles descartarías, razonando la respuesta.

> 1.3.2.3. Un profesor de la Escuela Politécnica Superior de la Universidad de Alicante quiere hacer un programa para facilitarle la corrección de prácticas. Como es una persona muy atareada, ha encargado a tres o cuatro ex-alumnos suyos que lo hagan en equipo. Vamos a suponer que, por ser "del gremio", sabe perfectamente lo que quiere de antemano, y que es un proyecto que no debería llevar demasiado tiempo al equipo de cuatro desarrolladores. A juzgar por esta información razona qué modelo de ciclo de vida de los vistos anteriormente es el más adecuado (o los más adecuados, si hay más de uno).

1.3.3. Metodologías de desarrollo software

Debido a que desarrollar productos software no es una tarea fácil, existen diferentes metodologías que podemos aplicar para dicho desarrollo. Entendemos por **metodología** un conjunto de técnicas y métodos que permiten abordar de forma homogénea cada actividad del ciclo de vida de un proyecto. Con esto se consigue:

- Optimizar el proceso y el producto software obtenido
- Obtener métodos que guían la planificación y el desarrollo
- Definir qué hacer, cómo hacerlo y cuándo, durante todo el proyecto

1.3.3.1. Elementos de una metodología

Una metodología se compone de:

- **Fases**: conjunto de actividades a realizar para obtener un subproducto o paso determinado dentro del proceso
- **Productos**: conjunto de entradas y salidas requeridas o producidas por cada fase.
- **Procedimientos y herramientas**: elementos que sirven de apoyo para realizar cada tarea, tales como editores de código, software de planificación, etc.
- **Criterios de evaluación** del proceso y del producto, para saber si se han logrado los objetivos

1.3.3.2. Tipos de metodologías

Algunas metodologías son más **tradicionales**, y se centran en controlar en todo momento el proceso y su planificación, estableciendo las actividades a realizar, los entregables que se deben producir en cada paso y las herramientas y notaciones que se emplearán. Se critica que, para ciertos proyectos, son metodologías rígidas y complejas de ajustar.

Otras metodologías se centran más en el factor personal y en el producto en sí. Son las metodologías **ágiles**, que valoran aspectos como la comunicación con el cliente, el desarrollo de software con iteraciones muy cortas, o la adaptabilidad a distintos tipos de proyectos. Son metodologías muy aceptadas para proyectos con requisitos cambiantes, o en los que el tiempo de desarrollo es un factor crítico.

La mayor parte de metodologías (tradicionales o ágiles) se basan en algunos de los modelos de ciclos de vida vistos con anterioridad. Algunos de los más utilizados son el modelo iterativo o incremental, y el modelo en espiral. Ambos se basan en las repeticiones de las diferentes etapas del software, con una serie de hitos intermedios, generando tras cada repetición un prototipo o versión del producto a desarrollar.

1.3.3.3. Ejemplos de metodologías tradicionales

Como hemos dicho, las metodologías tradicionales se centran en la planificación y la secuencia de pasos a seguir, por encima del producto a obtener o del propio cliente. Veremos algunas de las más representativas.

1.3.3.3.1. Rational Unified Process (RUP)

El proceso unificado Rational (RUP) es un marco de trabajo iterativo creado por Rational Software Corporation, una división de la empresa IBM. A diferencia de lo que las metodologías tradicionales suponen, RUP no es un marco rígido, sino adaptable a diferentes proyectos, seleccionando los elementos que sean más apropiados en cada caso.

La metodología RUP se basa en el modelo de ciclo de vida en espiral, tratando de paliar los evidentes defectos del modelo en cascada, e intentando dar cabida al paradigma orientado a objetos mediante UML (Lenguaje Unificado de Modelado), que veremos con más detalle en temas posteriores.

Módulos

RUP se estructura en una serie de elementos, denominados **módulos**. En concreto, los módulos principales de RUP son:

- **Roles**: definen conjuntos de competencias, habilidades y responsabilidades, de forma que la persona o personas con un rol específico, deben tener ese conjunto de cosas (competencias, habilidades y responsabilidades)
- **Productos de trabajo**: resultado de una tarea, incluyendo documentos y modelos secundarios producidos
- **Tareas**: unidad de trabajo, asignada a un rol, y que produce un producto de trabajo determinado y significante.

Estos tres módulos pueden interpretarse como el quién lo hace (rol), qué se hace (producto de trabajo) y cómo lo hace (tarea).

Fases

Según la metodología RUP, el ciclo de vida de un producto consta de cuatro fases:

- **Iniciación (*Inception*)**, donde se define el alcance del proyecto. En esta fase se realiza un modelado del negocio (análisis y documentación de la empresa para la que se trabaja,

viendo sus puntos fuertes y débiles, capacidad de generar beneficios, etc.), y un análisis de los requisitos del sistema a desarrollar.

- **Elaboración**, donde se analiza con mayor profundidad el proyecto y se define su arquitectura principal. Se realiza para ello un diseño y una implementación primarios que se toman como base. Esta fase, junto a la anterior, están orientadas a comprender el problema a resolver, definir la tecnología a utilizar y eliminar riesgos graves del proyecto.
- **Construcción**, donde se diseña la aplicación y se produce su código fuente. Se emplean para ello diferentes iteraciones, aplicando el modelo en espiral, y generando varios prototipos.
- **Transición**, donde se prepara el producto para su entrega al cliente

Podríamos pensar que esto se asemeja bastante al modelo en cascada, pero la esencia de RUP está en las iteraciones dentro de todas las fases (cada fase tiene un número variable de iteraciones, dependiendo del proyecto). Además, cada fase tiene un objetivo final que cumplir.

> Ejercicios propuestos

> 1.3.3.1. Busca en internet información sobre los tipos de diagramas que se integran en UML (*Lenguaje Unificado de Modelado*), que es una parte muy importante de la metodología RUP. Estos diagramas son de dos tipos: de estructura y de comportamiento. Indica al menos tres diagramas de cada uno de estos tipos.

1.3.3.3.2. Microsoft Solutions Framework (MSF)

MSF es otra metodología personalizable que, aplicando un enfoque tradicional, permite desarrollar un producto software. Como su propio nombre indica, es un enfoque ideado por Microsoft, y podemos aplicarlo no sólo a desarrollar aplicaciones, sino también a otros aspectos informáticos, como planificación de redes, infraestructuras, etc.

Fases

Al igual que RUP, MSF está basado en el modelo en espiral. El proceso de MSF consta de 5 fases o etapas iterativas, donde al final de cada fase se tienen unos objetivos a cumplir y entregables que completar.

- **Visión**: donde se valora el modelo de negocio, los beneficios que se pueden obtener, restricciones, alcances. Se debe obtener una especificación de requisitos, una valoración inicial de riesgos y una visión general de la empresa para la que se desarrolla. Es algo más o menos equivalente a la fase de iniciación del modelo RUP.
- **Planificación**: se planifica el desarrollo del proyecto y la arquitectura a utilizar, ajustándolo a un cronograma que cumpla con lo especificado. Se generan así la lista de actividades a completar, personas involucradas, responsabilidades y costes. Esto ayudará también a perfilar mejor los riesgos y evitar los que puedan resultar graves.
- **Desarrollo**: se comienza a construir el código a partir de las funcionalidades más básicas, entregando prototipos de cada una de ellas para someterla a pruebas y

evaluaciones por parte del cliente.
- **Estabilización**: se afina el producto final para que el cliente lo apruebe en su totalidad. Se documenta también el registro de pruebas realizadas previo a la aprobación del cliente.
- **Implantación y soporte**: se instala el sistema en el cliente, y se le ofrece garantía durante el tiempo estipulado en el contrato

Como principales ventajas están su adaptabilidad a distintos tipos de proyectos (al igual que RUP), la vinculación con el cliente, o la reutilización de elementos desarrollados en iteraciones anteriores. Como principales inconvenientes están la excesiva documentación a generar, y la dependencia en ocasiones de productos Microsoft (con el sobrecoste que esto puede conllevar).

1.3.3.4. Ejemplos de metodologías ágiles

Las metodologías tradicionales han demostrado ser muy válidas para proyectos grandes (en términos de tiempo y presupuesto), pero algunos aspectos, como la excesiva documentación a generar, y una posible sobreplanificación, las hacen a menudo inviables para proyectos pequeños, o de entrega rápida, o en los que los requisitos pueden cambiar con cierta frecuencia durante el desarrollo.

Ante este problema, muchos equipos de desarrollo tienden a prescindir de estos esquemas, y asumir metodologías ágiles, especialmente orientadas a proyectos pequeños (a desarrollar en poco tiempo, con equipos normalmente inferiores a 10 personas). En ellas se fomenta el trabajo en equipo y la responsabilidad propia, alineando el desarrollo con las necesidades del cliente y los objetivos de su compañía. La comunicación cara a cara entre los miembros del equipo, y con el cliente, suele ser un factor cotidiano. Los miembros del equipo se informan unos a otros de lo que hicieron la sesión anterior y de lo que van a hacer hoy.

Para cumplir con su cometido, el desarrollo ágil realiza el trabajo en incrementos pequeños, con planificación mínima. Cada incremento realiza una iteración de las etapas del software (requisitos, diseño, implementación, pruebas), en un espacio de tiempo pequeño (normalmente 1 – 4 semanas), conocido como *timebox*. De esta forma, se minimiza el riesgo general, y el proyecto se puede adaptar a cambios durante el desarrollo. La documentación se produce cuando se necesita, y la idea es tener tras cada iteración un prototipo que funcione, aunque sea con funcionalidades muy limitadas si aún no se han añadido suficientes requisitos.

1.3.3.4.1. El manifiesto ágil

El manifiesto ágil es una especificación creada en 2001 que recoge los valores y principios que debe tener una metodología de desarrollo para permitir a un equipo construir software rápidamente y respondiendo a los cambios que pueda haber en el futuro.

Algunos de los principios más importantes son:

- La gente es el principal factor de éxito de un proyecto. Es más importante construir un buen equipo y que éste configure su entorno de trabajo, que construir un entorno de trabajo y obligar al equipo a que se adapte a él.
- No se deben producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión
- Debe existir una interacción constante entre el cliente y el equipo de desarrollo
- La habilidad de responder a los cambios que pueda haber durante el desarrollo del proyecto (tanto en requisitos como en el propio equipo de trabajo, u otros factores) es más importante que ceñirse a un plan estricto y rígido
- La prioridad más alta es satisfacer al cliente mediante la entrega temprana y continua de software operativo
- La conversación cara a cara es el método más eficiente de hacer llegar la información en un equipo de desarrollo
- Simplicidad

1.3.3.4.2. Algunas prácticas habituales

Además de los principios recogidos en el manifiesto ágil, algunas metodologías ágiles aplican ciertas prácticas adicionales que son bastante habituales. Veremos algunas a continuación

Programación por parejas (PP)

La programación por parejas es una técnica de desarrollo que ofrece muchas ventajas. En esta práctica, dos programadores trabajan juntos en el mismo ordenador. Uno teclea el código (driver) y el otro revisa lo tecleado (observer). Ambos programadores cambian sus roles con frecuencia (30 minutos, por ejemplo). El observador, además, se encarga un poco de dirigir el trabajo, generando ideas para resolver futuros problemas, y haciendo que el driver se centre en completar su tarea actual, teniendo al observador como apoyo.

Entre las ventajas principales, se tienen:

- Programas más cortos, con menos errores y mejor diseñados (el código debe ser legible por los dos, y no sólo por uno).
- Es una práctica que sirve como aprendizaje y formación, si uno de los miembros de la pareja es experimentado y el otro novato, o si tienen conocimientos diferentes, se pueden transferir experiencia de uno a otro.
- Si algún miembro abandona el equipo, existe menos riesgo de retraso en las entregas u otros problemas de gestión, ya que el otro miembro de la pareja puede continuar el trabajo, apoyado por un nuevo miembro que se incorpore.
- Existen menos interrupciones en el proceso, ya que la alternancia de los compañeros hace que se pueda desarrollar el trabajo de forma más continuada.

Como inconvenientes, podemos citar:

- Algunos ingenieros prefieren trabajar solos

- Un desarrollador menos experto puede sentirse intimidado por trabajar con otro que lo sea más, y viceversa (los desarrolladores expertos pueden encontrar tedioso tener que ayudar o trabajar con alguien más novato).
- Coste (hay que pagar a dos personas por puesto de trabajo)
- Hábitos personales molestos de la pareja de programación

Desarrollo dirigido por las pruebas (TDD)

El desarrollo dirigido por las pruebas (*Test Driven Development*) es una técnica de desarrollo software que emplea iteraciones cortas basadas en casos de prueba escritos previamente. Así, cada iteración produce el código para que esas pruebas resulten satisfactorias, y una vez lo sean se integra el código con lo anterior y se refactoriza para optimizarlo. El ciclo de desarrollo propuesto por TDD es el siguiente:

1. Se añade una prueba al conjunto
2. Se ejecuta el conjunto de pruebas, verificando que todas funcionan menos la última que se ha añadido
3. Escribir código para resolver la última prueba añadida
4. Ejecutar de nuevo el conjunto de pruebas automatizadas y comprobar que todas funcionan correctamente
5. Refactorizar el código para mejorar su diseño
6. Volver al paso 1

Es importante disponer de un conjunto de pruebas unitarias de ejecución automatizada, de forma que poco a poco se van añadiendo más pruebas al conjunto, y en cualquier momento se puede lanzar el conjunto de pruebas y ver si todas se realizan satisfactoriamente. Veremos más adelante mecanismos de automatización de pruebas.

La idea es, por tanto, definir las pruebas que debe superar el sistema antes de escribir el código correspondiente, para así asegurarnos de que la aplicación va a poder ser probada, y de que se escriben pruebas para cada característica de la misma. Así evitamos escribir código innecesario (que no sirva para satisfacer ninguna prueba).

Propiedad colectiva del código

En muchas metodologías ágiles, cualquier programador del equipo puede cambiar cualquier parte del código en cualquier momento. De hecho, en algunas de las metodologías hacen que los distintos desarrolladores pasen por las diferentes partes del proyecto y las conozcan y sepan manipular, para prevenir posibles problemas si alguno de los miembros abandona el equipo.

Integración continua

La integración continua implica que cada pieza de código se integra en el sistema cuando está lista, y así el sistema se reconstruye cada vez (puede llegar a reconstruirse varias veces en un mismo día). Veremos más adelante algunas herramientas para integración de código y

automatización de la construcción. Estas herramientas construyen el software desde un repositorio de fuentes, y lo despliegan en un entorno de pruebas.

De esta forma, cuando un desarrollador debe realizar un cambio en el sistema, toma una copia del código actual, lo modifica y lo vuelve a subir al repositorio común, utilizando algún software o plataforma de control de versiones (Github, Bitbucket, CVS...). Esto puede plantear problemas si se tarda demasiado en actualizar los cambios, y otros miembros del equipo han modificado el código común demasiado como para que el programador "rezagado" pueda actualizar su parte nuevamente. Veremos esta problemática con más detalle en futuros temas.

1.3.3.4.3. Ejemplo: eXtreme Programming (XP)

La programación extrema (XP) es una metodología de desarrollo ágil introducida por Kent Beck. Se basa en la realimentación continua entre el cliente y el equipo de desarrollo, y en la simplicidad de las soluciones implementadas. Es especialmente adecuada (entre otros casos) para proyectos con requisitos muy cambiantes y alto riesgo técnico. Sus principios y prácticas son de sentido común, pero llevadas al extremo (de ahí su nombre).

Elementos

Los elementos que componen la metodología XP son:

- Las **historias de usuario**, técnica utilizada para especificar los requisitos del sistema a desarrollar. Son tarjetas de papel donde el cliente describe brevemente las características de dicho sistema (requisitos funcionales o no funcionales). Cada historia de usuario es lo suficientemente comprensible y delimitada como para que el equipo de desarrollo pueda implementarla en unas pocas semanas. Para ello, cada historia se descompone en tareas y se asignan a los programadores del equipo para implementarlas durante una iteración.
- Los **roles** de cada miembro del equipo. Encontraremos aquí:
    - **Programadores** que escriben el código y las pruebas del mismo
    - **Cliente** que escribe las historias de usuario y las pruebas funcionales para validar los productos entregados por el equipo. Además, asigna la prioridad a cada historia de usuario, y decide cuáles se implementan en cada iteración.
    - **Encargado de pruebas**, ayuda al cliente a escribir las pruebas funcionales, y ejecuta las pruebas regularmente, difundiendo los resultados al equipo
    - **Encargado de seguimiento**, verifica el acierto en las estimaciones de tiempo realizadas, proporcionando realimentación al equipo para estimar mejor las futuras iteraciones
    - **Consultor**, miembro externo al equipo con conocimiento específico en algún tema concreto sobre el que puedan surgir problemas. En algunos proyectos puede no ser necesaria esta figura.
    - **Gestor o *big boss***, vínculo entre clientes y equipos de desarrollo, se encarga de coordinar la comunicación entre ambas partes.

El proceso XP

El ciclo de desarrollo que utiliza la metodología XP sigue estos pasos:

1. El cliente define el valor de negocio a implementar (historia de usuario)
2. El programador estima el esfuerzo necesario para su implementación
3. El programador construye ese valor o historia
4. Se vuelve al paso 1

En cada iteración también actúan, en paralelo, los encargados de pruebas y seguimiento.

Prácticas

Para que la metodología XP cumpla correctamente su cometido, es necesario aplicar una serie de prácticas durante el proceso de desarrollo. Además de las recogidas en el manifiesto ágil (comunicación frecuente con el cliente, entregas pequeñas y frecuentes, diseño simple...), y de algunas prácticas comunes en muchas metodologías ágiles (programación por parejas, integración continua, propiedad colectiva del código...), podemos citar estas otras:

- Seguir ciertos **estándares de programación** por parte de todo el equipo, para mantener el código legible
- La **codificación** es el producto más importante del proceso, ya que sin el código no se tiene nada. De hecho, en XP se recomienda que, si se duda entre diferentes alternativas a la hora de implementar un problema, se implementen todas ellas y se valore cuál puede resultar mejor a partir del código generado.
- Las **pruebas** son la única vía de comprobar que el producto que se ha desarrollado hace lo que se pretendía
- Los programadores también deben **escuchar** al cliente y comprender su modelo de negocio (y no que se lo hagan llegar terceras personas), para así tener más preparación para desarrollar el producto que el cliente necesita.

> Ejercicios propuestos:

> 1.3.3.2. En este vídeo tienes un resumen de la metodología XP y de algunos conceptos generales de las metodologías ágiles. Después de verlo, responde a las siguientes preguntas:
>
> 1. Indica al menos 4 de los 12 principios que mencionan en el vídeo sobre el manifiesto ágil
> 2. ¿Para qué tipo de empresas y de proyectos está especialmente concebida la metodología XP?
> 3. Completa la frase: "Todo el software en XP se produce mediante la puesta a punto de..."
> 4. En XP, como en muchas metodologías ágiles, todos los desarrolladores tienen la misma jerarquía. En ese sentido, ¿qué papel juega el "jefe de proyecto" en XP?

1.3.3.4.4. Ejemplo: Scrum

Scrum es otra metodología ágil que puede usarse para desarrollar proyectos complejos, utilizando prácticas iterativas e incrementales. Puede aplicarse tanto a productos software como en otros ámbitos. Su nombre viene a significar algo así como la melé que se forma en rugby.

Roles

Los roles principales en Scrum son:

- **ScrumMaster**, sería algo así como el *jefe de proyecto*, aunque no existe tal figura en Scrum (el propio equipo se auto-organiza). Su principal misión es asegurarse de que el método Scrum se utiliza correctamente, y no se produce ninguna influencia externa que lo altere.
- **ProductOwner**, representante de la parte cliente, que no tiene por qué pertenecer a la empresa del cliente (puede ser alguien del equipo de desarrollo que actúe en nombre del cliente).
- **Team**, conjunto de desarrolladores

Proceso

Para empezar, y para hacerse una idea de la aplicación a desarrollar, se obtienen los requerimientos del cliente (tanto ejecutivos, directivos, como demás miembros de la empresa que vayan a emplear la aplicación). Estas características se toman mediante lo que se conoce como *historias de usuario*, en las que cada usuario preguntado cuenta qué espera que haga la aplicación. Estas historias se agrupan en una colección llamada *backlog* de producto. De ese *backlog*, se deben seleccionar qué historias se añadirán finalmente a la aplicación.

A partir de este punto se inicia el proceso de desarrollo, basado en iteraciones o *sprints*. En cada iteración (que normalmente dura entre 2 y 4 semanas, a elegir por el equipo de desarrollo), se crea un incremento o versión de software operativa. En ese incremento se añaden una serie de requisitos extraídos del *backlog*. El conjunto de requisitos a añadir en la iteración se decide en una reunión de planificación de cada iteración. En ella, el *Product Owner* informa al equipo de los ítems de ese backlog que se deben incorporar al producto, y el equipo determina cuántos de esos ítems se pueden añadir en la próxima iteración. Después, durante la iteración, el *backlog* se congela, no se pueden modificar los requisitos hasta la siguiente iteración.

Además de esta reunión al principio de cada iteración, en Scrum también se tienen reuniones diarias (donde se discute el estado del proyecto, lo que se ha hecho y lo que se va a hacer), y al final de cada iteración para evaluar lo que se ha obtenido.

Prácticas

Además de las prácticas comunes a toda metodología ágil, en Scrum podemos destacar estas otras:

- El cliente se convierte en parte del equipo de desarrollo

- Monitorización y gestión de riesgos frecuente en cada iteración
- Transparencia en el desarrollo. Todos deben saber quién es responsable de qué y cuándo, en todo momento.

> Ejercicios propuestos

> 1.3.3.3. En este vídeo tienes un resumen de la metodología Scrum. Después de verlo, responde a las siguientes preguntas:
>
> 1. ¿Qué es un *sprint* en Scrum?
> 2. ¿Quién es el encargado de priorizar las tareas a realizar y decidir qué se hace en cada *sprint*?
> 3. ¿Cómo se puede calcular la fecha de finalización y entrega de un *sprint*?

1.3.3.4.5. Ejemplo: DSDM

El Método de Desarrollo Dinámico de Sistemas (DSDM) es un enfoque ágil, iterativo e incremental que, como los anteriores, trata de entregar sistemas software en tiempo y presupuesto, ajustándose a los requisitos cambiantes del cliente durante el proceso. Está más enfocado a proyectos con planificación (plazos) y presupuestos estrictos.

Proceso

DSDM consta de tres fases:

- **Pre-proyecto**, donde se introduce el producto a desarrollar y sus requisitos
- **Ciclo de vida** del proyecto, que a su vez está subdividida en cinco etapas:
    - Estudio de viabilidad
    - Estudio de negocio
    - Iteración de modelo funcional
    - Iteración de diseño
    - Implementación
- **Post-proyecto**, donde se entrega el producto al cliente, se evalúa y se da un posible plan de mantenimiento posterior.

Prácticas

Además de las prácticas comunes a todas las metodologías ágiles, en DSDM podemos citar las siguientes:

- El equipo debe tener capacidad para tomar decisiones importantes para el progreso del proyecto, sin esperar aprobaciones de más alto nivel
- Todos los cambios realizados durante el desarrollo deben ser reversibles
- Las pruebas se realizan durante todo el ciclo de vida del proyecto
- Es preferible que los entregables traten las necesidades de negocio actuales, antes que hacer entregables más completos que traten otras posibles necesidades. En este sentido,

se parte de la idea de que el 80% de los beneficios del negocio vienen de un 20% de los requisitos, por lo que el equipo se centra en implementar ese 20% crítico primero. El 80% restante se puede añadir en sucesivas iteraciones, y así se mitiga el riesgo de salirse de plazos o presupuesto.