PRÓLOGO

¿A QUIÉN VA DIRIGIDO ESTE LIBRO?

La edición del ejemplar que tiene en sus manos ha sido fruto de cuatro años de trabajo y de investigación en el campo de la *Ingeniería del Software* y *del Lenguaje Unificado de Modelado* (UML). Como especialista en esta disciplina, y después de una larga trayectoria de desarrollos en diversos lenguajes de programación, he intentado aplicar todos mis conocimientos a esta obra, otorgándole cierto rigor académico y sobre todo un aspecto atractivo para el lector. Para ello me he servido de ejemplos basados en desarrollos multimedia, de sistemas y como no, de software de gestión. El presente libro no intenta ser una obra de referencia en el campo, sino más bien un libro de ayuda al estudiante y profesional de UML, con el firme propósito de transmitir la importancia que tiene la formación arquitectónica del software después de la algorítmica.

La lectura de los contenidos le será más amena si posee ciertos conocimientos a nivel de *Ingeniería Técnica* o *Ciclo Superior de FP en Informática*. De esta forma, las nociones de *Sistemas Operativos*, *Algorítmica*, *Redes*, *Bases de Datos*, *Autómatas* y *Programación Orientada a Objetos* serán de una valiosa utilidad para seguir las indicaciones dadas en los capítulos.

Finalmente, si usted posee conocimientos avanzados de los lenguajes Java, C++ y/o Python, le resultará más fácil el aprendizaje de los conceptos aquí explicados. En caso contrario le podrá ser de utilidad el anexo A y las referencias bibliográficas sobre el tema que podrá encontrar al final del volumen. Es por tanto importante que revise estos contenidos antes de comenzar a leer.

ESTRUCTURA DEL LIBRO

Los contenidos de la presente obra se han estructurado de acuerdo a un ciclo de vida secuencial de un desarrollo software. La sucesión de las explicaciones teóricas de los capítulos está acordes al avance de un proyecto real.

Con la idea de explicar la asociación existente entre UML y un desarrollo real, se presentan tres proyectos que van evolucionando al ritmo del ciclo de vida, es decir, desde la adquisición de requisitos hasta la implementación. Dichos ejemplos nos irán guiando en los diferentes diagramas UML del proyecto y sus fases de construcción. De esta forma el libro se ha estructurado en el siguiente orden metodológico:

- Parte I: Introducción teórica.
- Parte II: Análisis de requisitos.
- Parte III: Diseño arquitectónico.
- Parte IV: Diseño detallado.
- Parte V: Implementación.

Se ha intentado no descuidar los aspectos teóricos ni los prácticos, pretendiendo así conjugar a lo largo del libro ambos matices.

NOVEDADES EN LA SEGUNDA EDICIÓN

Se presenta aquí la segunda edición de la obra y, como es lógico en una nueva edición, se han corregido las erratas de la primera edición y se ha actualizado el contenido con las últimas tendencias en *Ingeniería de Software* y lenguajes de programación más demandados en la industria. Por esa razón, en el primer capítulo hallará una ampliación con el ciclo de vida incremental, metodologías de desarrollo software, calidad para la normalización/estandarización, así como teoría sobre el desarrollo de software seguro. También se ha incluido un nuevo patrón de diseño GOF y se ha tenido en consideración, por su importancia, incluir un nuevo capítulo dedicado a patrones GRASP con el fin de dar a conocer las técnicas que facilitan buenas prácticas de programación orientada a objetos actualmente.

La gran novedad de la presente edición recae en la incorporación de un nuevo caso de estudio para el abordaje de los conocimientos sobre análisis, diseño y programación mediante el lenguaje multiplataforma y multiparadigma Python, debido principalmente a su éxito en el mundo empresarial y de investigación.

Por todo ello, espero que esta obra cubra con completitud los conocimientos teóricos y prácticos que todo desarrollador aspira a alcanzar en aras de la construcción de software complejo que tanto necesitan las compañías y los investigadores de todo

el mundo. Saber que este libro le ha servido para lograr el citado propósito sería una de mis grandes satisfacciones.

ESTILOS DE PROGRAMACIÓN

Como corresponde a cualquier proyecto informático basado en programación, un buen código fuente no es solo el que realiza sus funciones correctamente, sino también el que es legible y mantenible a lo largo del tiempo. Por este motivo, en el libro se ha intentado seguir ciertos criterios estilísticos en relación a cada lenguaje. Así para el código Java se ha utilizado las guías de estilo oficiales de *Sun/Oracle* publicadas en su sitio Web, mientras que para C++ el estándar elegido ha sido el *Joint Strike Fighter - Air Vehicle - C++ Coding Standards* recomendado por *Bjarne Stroustrup*. Respecto a Python, se ha aplicado el estilo de codificación indicado en su página de referencia: www.python.org.

DIAGRAMAS Y PROGRAMAS DE EJEMPLO

Para diseñar los diagramas de ejemplo del libro se han utilizado las aplicaciones *StarUML* y *MagicDraw* 15.0. La primera de ellas es una aplicación gratuita escrita inicialmente en Object-Pascal (*Borland Delphi*) y posteriormente en Java/Eclipse que puede ser descargada en la siguiente URL: http://staruml.sourceforge.net. La segunda es una aplicación comercial, pero puede descargarse una versión de prueba en: http://www.nomagic.com con el fin de evaluar los proyectos propuestos en el libro.

Las aplicaciones y ejemplos de Java aquí descritos han sido compilados y ejecutados directamente desde la línea de comandos en Windows, mientras que los ejemplos de C++ han sido programados sobre Visual C++ 2017. Si está interesado en la última actualización, puede descargarse una versión gratuita desde la Web de Microsoft en https://visualstudio.microsoft.com/es/downloads/ En cuanto a los proyectos realizados en Python, usted podrá descargar el intérprete del lenguaje desde su sitio web: https://www.python.org/downloads/

Finalmente, si desea descargar tanto los diagramas UML como los códigos fuente debe acceder a la web del propio libro en: www.ra-ma.com

Le recomiendo que lea primeramente el fichero *leame.txt* que se encuentra en el directorio con las indicaciones. Este fichero le guiará en la instalación de los ejemplos y le informará de la estructura de los archivos.

FEEDBACK

Cualquier crítica constructiva será siempre bienvenida. Éstas me servirán para mejorar mi trabajo de cara a una futura edición; por lo que cualquier errata, error, inconsistencia o fallo de presentación serán bien recibidas por mi parte.

Para cualquier consulta o informe de fallos puede contactarme por e-mail: cjimeneztau@gmail.com

AGRADECIMIENTOS ACADÉMICOS

Quisiera expresar mi agradecimiento al profesor *Dr. Manuel Arias Calleja* por su amabilidad al revisar y dirigir algunos contenidos vía e-mail y telefónica desde Madrid; al *Departamento de Ingeniería de Software y Sistemas Informáticos* de la UNED por los conocimientos transmitidos en su Máster en Investigación y al profesor *Dr. Jesús García Molina*, de la *Universidad de Murcia*, por su segunda revisión y sabios consejos.

Murcia

El autor

Enero de 2021.

1

UML EN EL CONTEXTO DE LA INGENIERÍA DEL SOFTWARE

«Siempre imaginé que el Paraíso sería algún tipo de biblioteca».

(Jorge Luis Borges)

Comenzamos la explicación de los conceptos relacionados con el mundo de la construcción del software haciendo una breve síntesis de la historia y evolución de las técnicas de modelado y cómo ha sido posible la llegada de UML (*Unified Modeling Language*) al mundo de la IS (*Ingeniería del Software*). En este capítulo también se hará una reseña a los diferentes diagramas que se tratarán a lo largo del libro y la relación que tienen con los diferentes ciclos de vida del software. Para finalizar se introducirá la última tendencia en el desarrollo automático de software mediante modelado con las técnicas de MDA (*Model Driven Architecture*).

1.1 LA NECESIDAD DE UN MODELO UNIFICADO

UML comienza a gestarse cuando la empresa de software *Rational*, fundada por *Grady Booch,* contrata a *James Rumbaugh* en 1994 que por entonces trabajaba en la *General Electric.* UML nació como la fusión de la metodología OMT (*Object-Modeling Technique*) de *Rumbaugh* y el método de *Grady Booch*. Al poco tiempo (1995) se unió a Rational *Ivar Jacobson*, inventor del método OOSE (*Object-Oriented Software Engineering*) y de algunos conceptos de otros lenguajes de modelado. Al grupo de los tres inventores se le llamó familiarmente como los "*Tres Amigos*" a causa de sus frecuentes debates sobre UML. En enero de 1997 fue propuesto el

primer borrador de UML 1.0 a la OMG[1] (*Object-Management Group*) a través de un consorcio llamado *UML Partners*. Más tarde, en noviembre de 1997, y después de la creación de la *Semantic Task Force* por *UML Partners* para estandarizar UML y dotarla de contenido semántico, UML 1.1 fue adoptada por la OMG. Desde la versión 1.1 hasta hoy han existido varias revisiones menores de la especificación como la 1.2, 1.3 y 1.4. La versión 2.5.1 de UML, lanzada en diciembre de 2017, es la última a fecha de escritura de la presente edición.

Actualmente UML goza de un reconocido prestigio en el campo de la IS, aplicándose de forma extensa en una gran variedad de campos del desarrollo de software (incluidos la Inteligencia Artificial) y con un éxito sin precedentes en los proyectos de software. La versión de UML utilizada en este libro es la 2.x y data de julio 2005.

1.2 ¿POR QUÉ MODELAR?

Un modelo es una abstracción de un problema de la realidad. Con esta idea surge el concepto de modelar, que consiste en abstraer las características esenciales de un problema real a una representación útil para un propósito determinado. En el caso de la ingeniería convencional como la aeronáutica o la mecánica, los ingenieros construyen modelos para asegurarse que el producto final funcionará. La implicación directa del modelo es que es posible su validación y comprobación, por lo que no tiene sentido construir modelos para luego no realizar pruebas. Obviamente los modelos son más baratos que los productos acabados y además permiten verificar su correcto funcionamiento.

En el mundo del software, el ingeniero construye modelos para probar si sus proyectos tendrán éxito y para comunicar a otros ingenieros y programadores las ideas de lo que intenta modelar. En el campo de la IS predomina el uso de UML y no sería lógico utilizar otra técnica para organizar la estructura estática o dinámica de una aplicación, ya que el lenguaje de modelado UML es bastante maduro para este fin. Con el uso de UML el ingeniero puede crear un modelo más factible e inteligible que el código fuente complejo, intercambiable entre expertos y que permite probar fácilmente la aplicación. Las pruebas desde UML son un campo todavía en experimentación al que se dirige la tecnología MDA (*Model Driven Architecture*) y que aún no es fidedigna del todo. Actualmente ya es posible la conversión directa

1 La OMG es un consorcio sin ánimo de lucro formado por varias empresas tecnológicas. Su finalidad es la gestión y estandarización de tecnologías orientadas a objetos como metodologías, bases de datos, CORBA, etc.

de un modelo UML a una aplicación ejecutable y distribuible, aunque cuenta con la desventaja del excesivo coste económico de estas herramientas.

Es siempre más factible dedicar un tiempo prudencial a analizar y diseñar la aplicación para comprobar que funciona, ya que el hecho de realizar esta acción nos asegura evitar muchos errores de diseño y la construcción de programas erróneos. En este sentido, nuestro trabajo siempre podrá ser entendido, discutido y comunicado con otros colegas de otras compañías o nacionalidades de una manera estándar, lo que implica también una forma de documentar productos de software para terceros desarrolladores.

En resumen, UML no se utilizará para probar si nuestro programa funcionará correctamente, sino para discutir con otros, documentar, aplicar la ingeniería directa o simplemente representar una idea. Por último, UML no es adecuado para sistemas en tiempo real, para tales sistemas existen notaciones específicas que no están dentro del ámbito de este libro.

1.3 MODELOS DE PROCESO SOFTWARE

De igual forma que sucede en otras actividades de ingeniería, en la que existe un ciclo de vida desde que se detecta la necesidad de construir el producto hasta que está en uso, la IS también requiere de tiempo y esfuerzo para su desarrollo y debe permanecer en uso durante un tiempo mucho mayor. Por este motivo se requiere de un ciclo de construcción y mantenimiento del software que se repita de forma secuencial o de otras maneras y permita el establecimiento de fases de desarrollo. Al conjunto de fases delimitadas con sus entradas y salidas se les denomina comúnmente “ciclo de vida” y permiten la elaboración de software de una manera metódica y ordenada.

Las fases principales son comunes en todos los ciclos de vida y se repiten de manera secuencial, incremental o en espiral. Las fases principales en cualquier ciclo de vida son:

1. *Análisis*: Es el proceso de reunión de requisitos para construir el software. El analista del software debe comprender el dominio de información del software y construir el modelo de análisis.
2. *Diseño*: Esta fase se compone de muchos pasos en los que se encuentran la deducción de estructuras de datos, la arquitectura del software, la interfaz de usuario y el diseño algorítmico. Algunos autores dividen esta etapa en el *diseño global o arquitectónico* y *diseño detallado*. El primero consiste en transformar los requisitos en una arquitectura de alto nivel, definir las pruebas, generar la documentación y planificar la integración. El diseño

detallado consiste en refinar el diseño para cada módulo, definiendo sus requisitos y la documentación.

3. *Codificación*: Se transforma el diseño en código ejecutable para la construcción del sistema. Las herramientas Computer-Aided Software Engineering (CASE) actuales permiten la conversión de un modelo UML para generar una parte esquemática del código de la aplicación, aunque no toda completamente.
4. *Pruebas*: Después de generar el código se procede a probar el programa. El proceso de pruebas consiste en el análisis de los procesos lógicos internos del software (comprobando que las sentencias y las bifurcaciones se cumplen), o en el análisis de los procesos externos funcionales (se ejecuta el programa con una serie de entradas y condiciones para comprobar que se ejecuta correctamente). A las primeras se les denomina pruebas de "caja blanca", mientras que a las últimas se les denomina pruebas de "caja negra", por la opacidad de la visión del código fuente.
5. *Mantenimiento*: Se produce después de entregar el producto al cliente. Es la parte que más tiempo consume. Se debe comprobar que el software sigue funcionando correctamente, y que se adapta a los nuevos requisitos y condiciones externas. Por ejemplo, un cambio de sistema operativo o de un dispositivo o periférico.

Estas etapas o fases se dividen en tareas. La *documentación* es una tarea importante que se realiza en todas las fases y donde UML cumple un papel excelente para los desarrolladores.

1.3.1 Modelo en cascada

El ciclo de vida en cascada fue propuesto por *Winston W. Royce* en 1970 y posteriormente revisada por *Barry Boehm* en 1980 e *Ian Sommerville* en 1985. Se basa en una serie de etapas o fases que se ejecutan en el proyecto de forma iterativa. Es decir, partiendo del *análisis de requisitos* se transita por el resto de las fases secuencialmente hasta llegar al *mantenimiento*. Una característica de este ciclo de vida es que podemos volver hacia atrás en el ciclo si tuviéramos que realizar cambios en algunas de las etapas anteriores. Por ejemplo, si en la etapa de *pruebas* fuera necesario volver al *diseño*, tendríamos obligatoriamente que pasar de nuevo por la *codificación*.

Entre las ventajas de este modelo destacan la facilidad de utilización y la existencia de una gran cantidad de herramientas CASE que lo soportan. Entre las desventajas se incluye la exigencia de tener todos los requisitos al comienzo del

proyecto y que no genera ningún producto hasta que se hayan finalizado todas las fases (ciclo). En general, este tipo de modelo se utilizará en proyectos de corta duración y fáciles que no requieran de cambios frecuentes en los requisitos.

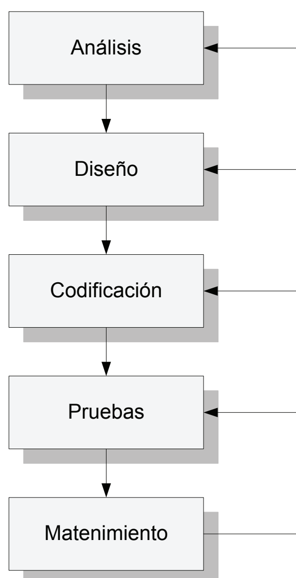

Figura 1.1. Ciclo de vida en cascada

1.3.2 Modelo incremental

Según [Pressman02], el modelo incremental corresponde a la gama de modelos evolutivos de proceso de software, ya que combina las ventajas del modelo en cascada con la filosofía de construcción de prototipos. Como veremos a continuación, la metodología RUP (Rational Unified Process) utilizará un esquema de ciclo de vida muy similar. En la figura 1.2 se muestra el diagrama de vida del citado modelo donde se puede apreciar los diferentes incrementos que producirán versiones incompletas del producto basadas en los incrementos anteriores, hasta llegar a una versión completa y estable. Cada incremento permite al usuario evaluar sobre un producto parcial y operacional las funcionalidades requeridas, de forma que facilite el desarrollo progresivo en caso de insuficiencia de personal con vistas a una fecha límite de entrega estricta. Una de las ventajas que se adquiere utilizando este método es la no necesidad de tener un conjunto completo de requisitos al principio del proyecto; no obstante, la detección de errores en este modelo de proceso se produce tarde [Arias07].

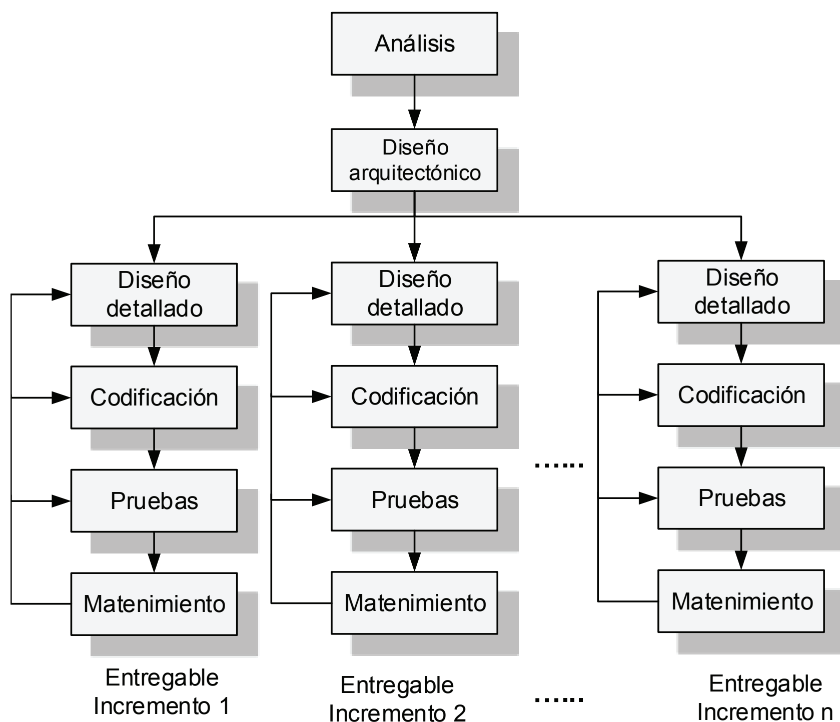

Figura 1.2. Ciclo de vida incremental

1.3.3 Modelo en espiral

Este ciclo de vida fue propuesto inicialmente por *Barry Boehm* en 1986 y consiste en una serie de ciclos con hitos que se repiten un número determinado de veces propiciando una evolución del producto hasta su conclusión final. Una característica importante de este ciclo de vida es la introducción del análisis del riesgo en cada una de las evoluciones, intentando optar por el más asumible.

Entre las ventajas de este modelo se incluye la capacidad para reducir riesgos, mejorar la calidad y la no necesidad de tener todos los requisitos al principio. Entre las desventajas se incluye la dificultad de adopción de riesgos y su coste. Sin embargo, debido a su naturaleza evolutiva y recurrente permite ser utilizado en proyectos complejos, críticos y de larga duración.

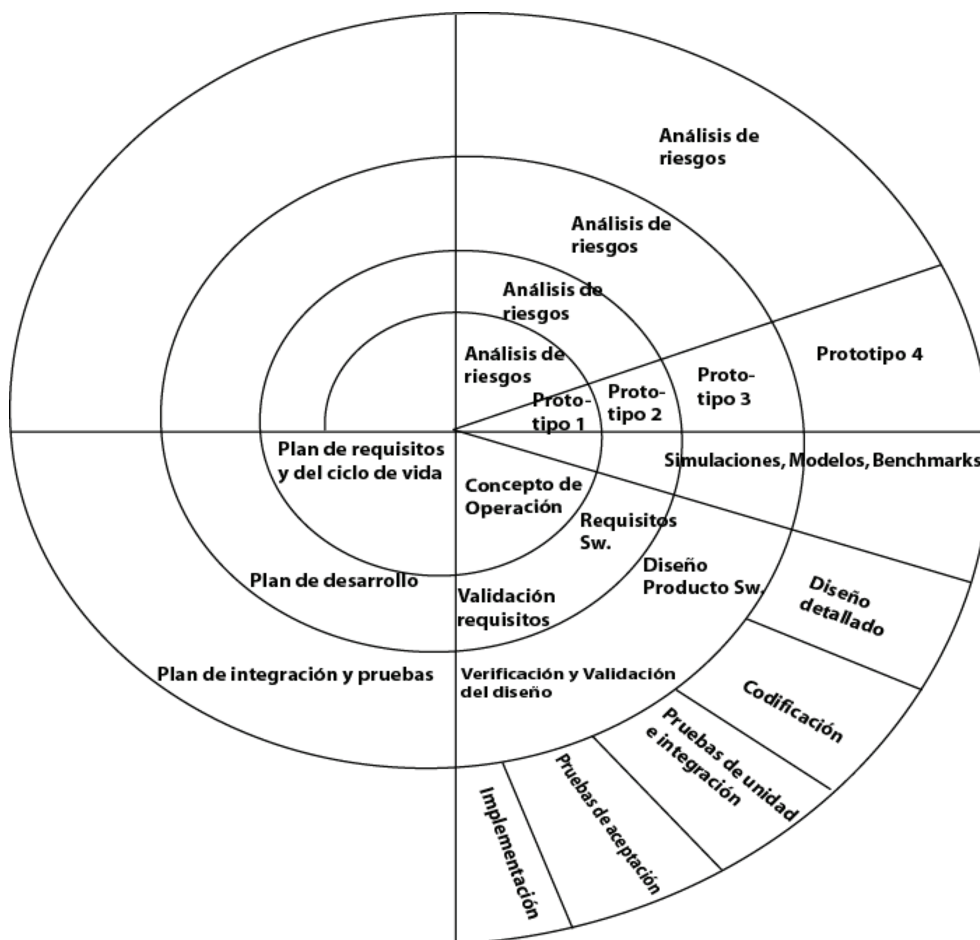

Figura 1.3. Ciclo de vida en espiral

1.3.4 Relación de las fases con los diagramas UML

En lo que respecta a los diferentes tipos de diagramas relacionados con el contenido del presente libro, se han estudiado aquellos que están involucrados en la parte de construcción de alto nivel donde se aplica la visión del ingeniero de software, es decir, los diagramas de representación de requisitos ya elicitados, análisis, diseño arquitectónico y diseño detallado. Cada diagrama se usa en una o más fases específicas del ciclo de vida.

Para comprender mejor esta relación se expone a continuación el siguiente esquema donde se desglosan los diferentes tipos de diagramas en relación a su fase dentro del ciclo de vida:

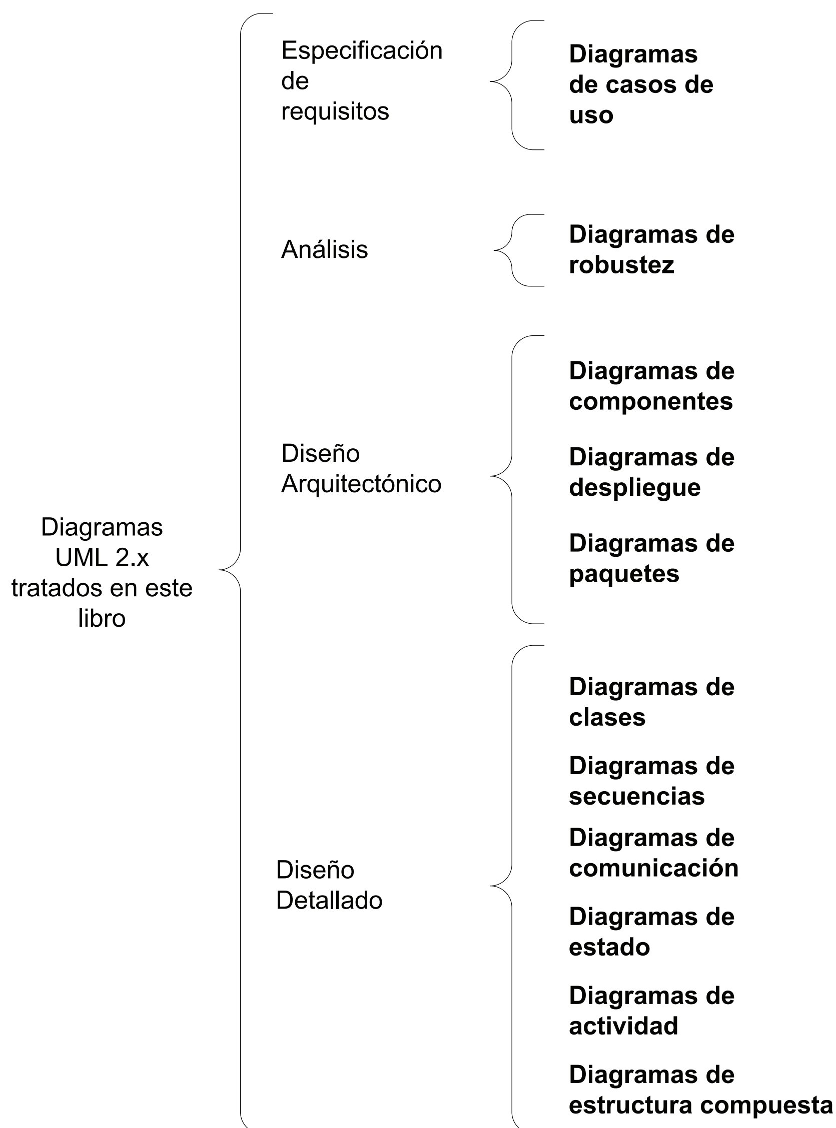

Figura 1.4. Esquema de los diferentes diagramas UML

En lo que respecta a la fase de Especificación de requisitos, nos encontramos con el *Diagrama de casos de uso***,** que permiten representar el comportamiento del sistema desde el punto de vista del usuario y cómo éste interactúa con el sistema para llevar a cabo los requerimientos de la aplicación. En la fase de Análisis se hallan los *Diagramas de robustez* con el fin de analizar los pasos de un caso de uso para validar la lógica de negocio, en otras palabras –permiten comprobar la robustez de un caso de uso y su conformidad con los requerimiento del sistema en construcción–.

Dentro de la fase de *Diseño arquitectónico* hallamos los *Diagramas de componentes* que son los responsables de modelar la visión estática del sistema en el nivel de implementación. Muestra la organización y las dependencias entre componentes software: librerías, *frameworks***,** interfaces de usuario, ficheros de cabecera, etc. También, y dentro de esta fase, se sitúa el *Diagrama de despliegue* con la idea de representar la organización del hardware y los principales artefactos de nuestro sistema. En el *Diagrama de paquetes* se muestra la división del sistema en unidades lógicas y muestra la dependencia entre ellas. Normalmente el paquete está representado como un directorio y es de gran utilidad especialmente en la organización de la aplicación tanto en C++ como en Java y Python[2].

En la fase de *Diseño detallado* es donde se ubica el *Diagrama de clases* que permite representar estáticamente el sistema, indicando la relación entre las diferentes clases, su estructura jerárquica y cómo se organizan. Seguidamente estarán los *Diagramas de interacción* que permiten visualizar detalladamente cómo los diferentes objetos involucrados en el sistema interactúan para ejecutar una tarea. La aplicación principal de los diagramas de interacción es mostrar el funcionamiento de un caso de uso y los pasos para completarlo. Por ello, en los *Diagramas de secuencias* y *comunicación* se modelan las interacciones entre los objetos una vez han sido instanciados en el sistema.

Los *Diagramas de estados* son los que representan la vista dinámica del sistema en cuanto que modelan la evolución de un sistema a lo largo del tiempo. En UML representan el comportamiento de las instancias de las clases, componentes o casos de uso. De forma similar se encuentran los *Diagramas de actividad*, que están basados en las *redes de Petri* y modelan fundamentalmente el paralelismo y el comportamiento del sistema haciendo énfasis en la participación de los objetos en ese comportamiento. Finalmente, los *Diagramas de estructura compuesta* permiten representar una visión estructural de los principales clasificadores utilizados en el modelo estático del sistema.

2 En estos dos últimos se le denomina paquete.

Como hemos podido observar, estos diagramas que se han explicado brevemente son de importancia capital en cualquier proyecto software basado en alguna metodología. A lo largo de los siguientes capítulos trataremos cada uno de ellos de forma más pausada.

1.4 METODOLOGÍAS DE DESARROLLO

Por *metodología de desarrollo de software* se entiende como el conjunto de roles o personal implicado en las fases de construcción del producto, los artefactos o recursos y materiales utilizados en el desarrollo del mismo y el ciclo de vida concreto utilizado. En esta sección se expondrán las dos formas de abordar una metodología de desarrollo actualmente, la cual puede ser ligera (ágil) o pesada.

1.4.1 Metodologías ágiles

De acuerdo a [Pressman14], una metodología ágil promueve la satisfacción del cliente como resultado de las prontas entregas incrementales de software gracias a equipos humanos muy motivados que aplican un trabajo ingeniería de software simple y ligero. No obstante, no todas las metodologías ágiles pueden ser aplicadas a todos los tipos de proyectos y escenarios. El auge y éxito de las metodologías ágiles actualmente es uno de los factores importantes a tener en cuenta en el momento de crear una empresa de desarrollo de software.

Por *ágiles* entenderemos metodologías *ligeras* que son fácilmente adaptables a la dinámica de los negocios y a los cambios, fomentando la comunicación entre clientes y desarrolladores por medio de grupos de trabajo con dotes comunicativas y bien motivados. La experiencia de décadas de desarrollo confirma que el coste de utilizar una metodología tradicional evoluciona casi exponencialmente con el tiempo; mientras que una metodología ágil lo hace "idealmente" de forma logarítmica. Por ello, las claves de una metodología ágil residen en ser *adaptables* a los imprevistos relacionados con los requerimientos del cliente y a los imprevistos del análisis, diseño, la implementación y pruebas. Por ello, un método ágil debe adaptarse incrementalmente a estas contingencias por medio de equipos que mantengan una relación constante con el cliente y realicen entregas operativas de prototipos en periodos cortos de tiempo.

1.4.1.1 EXTREME PROGRAMMING (XP)

Extreme Programming (XP) es un conjunto de principios de desarrollo descrito por *Kent Beck* en su libro sobre esta materia [Beck99]; por tanto, no es exactamente una metodología, sino un conjunto de principios que derivan en una

serie de prácticas. En la actualidad ha perdido importancia, aunque suele tener presencia frente a metodologías clásicas debido a su simplicidad e interacción con el cliente. Según [Pressman14], [Arias07] las principales actividades que lo componen (ver figura 1.5) son:

1. **Planificación**: Se crean las *historias de usuario* que son unos documentos cortos escritos por los usuarios pero sin vocabulario técnico. Cada historia especifica un requisito del sistema y debe durar "idealmente" entre una y tres semanas, conociendo de antemano que nuevas historias pueden ser escritas en cualquier momento. El equipo XP colabora conjuntamente con los clientes ordenando las diferentes historias en categorías dependiendo de su urgencia y riesgo. El cliente puede añadir historias, modificarlas, dividirlas o eliminarlas definitivamente a su gusto. Una desventaja de este modelo es que hay que tener disponible al cliente desde el comienzo al fin del desarrollo.

2. **Diseño**: Siguiendo el principio de ser lo más simple posible, se realizará el diseño evitando aproximaciones demasiado complejas. En esta actividad se utilizarán las tarjetas CRC (Clase, Responsabilidades, Colaboraciones)[3] y se producirá un prototipo operacional correspondiente al incremento del diseño. Además, XP se basa en la utilización de *refactorización* ("refactoring") y otras técnicas adicionales que no se comentarán aquí y que permiten cambiar la estructura interna del código fuente sin afectar a la funcionalidad o comportamiento externo de la aplicación.

3. **Implementación**: Una vez que las historias de usuario se han obtenido y el diseño preliminar ha finalizado, el equipo de desarrollo prepara las *pruebas de unidad*. Una vez que las pruebas de unidad han sido creadas, el programador puede enfocar mejor su atención sobre lo que debe ser implementado para pasar las pruebas. Cuando el código se ha implementado pueden realizarse las pruebas de unidad. Una de las características de XP es la programación por pares ("pair-programming"), ya que es una práctica recomendada, en la que un programador escribe código y el otro lo prueba y después se intercambian los papeles. Una vez que cada pareja ha finalizado, su trabajo puede ser integrado con el trabajo de otros equipos. XP garantiza una integración diaria por un equipo especializado. Una noción importante en XP es que el diseño y la implementación están a menudo entrelazados.

---

3 Cada tarjeta representa un objeto. En la cabecera se escribe el nombre de la clase que instancia, en la parte izquierda se indican las responsabilidades y, en la derecha, las clases con las que colabora junto a cada responsabilidad.

4. **Pruebas**: Las pruebas de unidad se automatizan y se aplican en esta fase, lo que facilita las *pruebas de regresión*. Las pruebas de validación y la integración son frecuentes durante esta actividad. Finalmente se realizan los *test de aceptación* que son realizados por los clientes sobre el sistema y se basan en las historias de usuario creadas al comienzo.

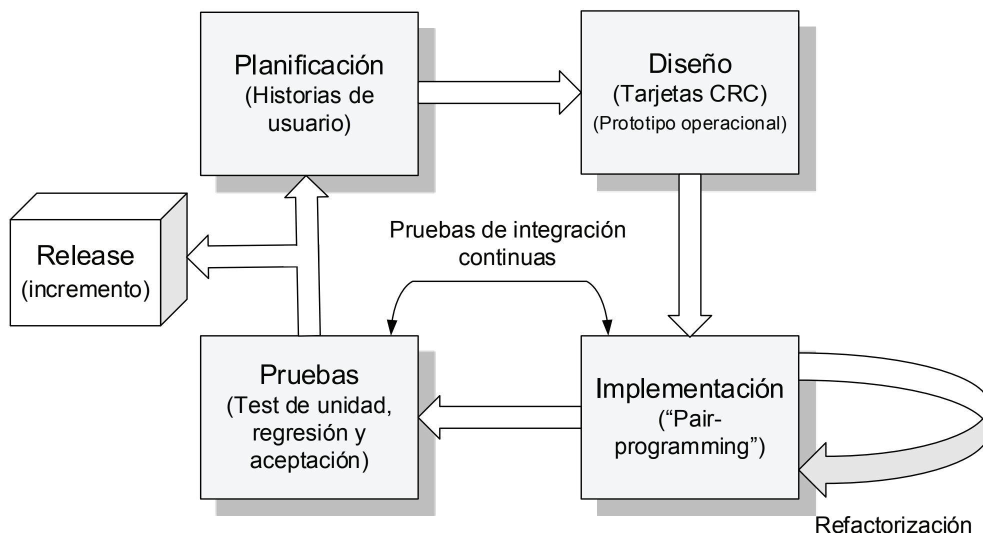

Figura 1.5. Metodología Extreme Programming (XP)

1.4.1.2 SCRUM

Scrum es otra metodología ágil utilizada ampliamente para productos complejos que comprende desde software y hardware hasta vehículos autónomos, escuelas y un amplio rango de organizaciones. Fue creada a comienzos de 1990 por Jeff Sutherland y Ken Schwaber [Sutherland17] con el propósito de definir un marco de trabajo para tratar con la complejidad. Scrum está compuesto por los siguientes elementos relacionados:

- Roles (*Roles*)
- Eventos (*Events*)
- Artefactos (*Artifacts*)
- Reglas (*Rules*), que relacionan a los anteriores elementos.

Equipo Scrum

El equipo scrum *(Scrum Team*) está compuesto por el Propietario del Producto (*Product Owner*), el Equipo de Desarrollo (*Development Team*) y un Scrum Master.

- **Propietario del Producto (*Product Owner*):** Es la persona encargada de gestionar la Pila del Producto (*Product Backlog*) de forma que realice acciones de ordenación de elementos para alcanzar los objetivos, optimizar el valor del trabajo que realiza el Equipo de Desarrollo. Adicionalmente, toda la organización debe respetar sus decisiones.
- **Equipo de Desarrollo (*Development Team*):** Se compone de profesionales autoorganizados y multifuncionales encargados de realizar un incremento del producto. Es importante la sinergia que resulte del equipo ya que ayuda a optimizar su eficiencia y efectividad. El tamaño ideal del Equipo de Desarrollo es de menos de tres miembros, puesto que reduce la interacción y resulta en ganancias de productividad más pequeñas. No es aconsejable equipos de más de nueve miembros por requerir demasiada coordinación.
- **Scrum Master (*Scrum Master*):** Es el encargado de promocionar y apoyar la Guía de Scrum [Sutherland17]. El Scrum Master es un líder que está al servicio del Equipo Scrum, en concreto del Propietario del producto, del Equipo de Desarrollo y de la Organización.

Eventos Scrum

En Scrum existen diferentes tipos de eventos que se definen como períodos de tiempo con una duración máxima.

- **Sprint (*Sprint*):** Es el corazón de Scrum y consiste en un período de tiempo (time-box) de un mes de duración durante el cual se crea un incremento de producto "Terminado" (*Done*)[4], utilizable y potencialmente desplegable.
- **Planificación del Sprint (*Sprint Planning*):** Se decide el trabajo a realizar durante el Sprint (objetivos, elementos de la Pila del Producto, etc.) y tiene una duración máxima de ocho horas para un Sprint de un mes. El Sprint puede ser cancelado, aunque no es recomendable debido a que consume recursos al reagruparse estos en otra planificación de Sprint y por tanto suele ser traumático.

4 El equipo Scrum debe tener un entendimiento compartido de lo que significa que el trabajo esté completado con el fin de asegurar la transparencia.

- **Scrum Diario (*Daily Scrum*):** Consiste en una reunión diaria con una duración de tiempo de quince minutos para el Equipo de Desarrollo donde se planea el trabajo de las siguientes veinticuatro horas, lo que optimiza la colaboración y el desempeño del equipo.
- **Revisión del Sprint (*Sprint Review*):** Es una reunión informal de no más de cuatro horas para un Sprint de un mes que se realiza a su término, con el fin de revisar el incremento y adaptar la Pila del Producto, lo que facilita la retroalimentación de información y el fomento de la colaboración.
- **Retrospectiva del Sprint (*Sprint Retrospective*):** Se trata de una reunión de como máximo tres horas para un Sprint de un mes en el que se inspecciona el último Sprint en cuanto a personas, relaciones, procesos y herramientas. Se identifican y ordenan los elementos más importantes y se crea un plan para implementar las mejoras de forma que el Equipo Scrum desempeñe su trabajo.

Artefactos

Representan el trabajo en diversas formas útiles en favor de la transparencia y oportunidades para la inspección y adaptación.

- **Pila del Producto (*Product Backlog*):** Contiene todo lo conocido que podría ser necesario en el producto y los requisitos (enumera características, funcionalidades, mejoras y correcciones que constituyen cambios a realizar sobre entregas futuras del producto). Es un artefacto dinámico que crece y se adapta para que el producto sea competitivo y útil. Su existencia está supeditada al ciclo de vida del producto.
- **Pila del Sprint (*Sprint Backlog*):** Este artefacto incluye los elementos de la Pila del Producto utilizados en el Sprint, junto con un plan para entregar el Incremento de producto y conseguir el objetivo del Sprint. Es una predicción hecha por el Equipo de Desarrollo que contiene la funcionalidad del siguiente Incremento y el trabajo necesario para entregar dicha funcionalidad al finalizar el Incremento.
- **Incremento (*Increment*):** Es el conjunto de todos los elementos de la Pila del Producto realizados durante el Sprint y los Sprint anteriores y supone un paso hacia una meta. Cuando se finaliza un Sprint este debe hallarse en el estado "Terminado" (*Done*) y en condiciones de utilizarse por si se decide liberarlo.

1.4.2 Metodologías pesadas

1.4.2.1 RATIONAL UNIFIED PROCESS (RUP)

Según [Arias07] y [Jacobson00], el Proceso Unificado de Rational es una metodología orientada a objetos desarrollada por los mismos creadores de UML. Las características de RUP son las siguientes:

- **Dirigido por casos de uso:** Representan los requisitos funcionales del sistema desde el punto de vista del usuario y servirán como guía a RUP durante su ciclo de vida. Los casos de uso se explicarán en el Capítulo 2.
- **Ciclo de vida iterativo e incremental:** El proyecto se divide en mini-proyectos que representarán una iteración.
- **Estructura del ciclo de vida:** El proceso RUP consiste en una serie de ciclos, donde cada ciclo se compone de las siguientes fases: inicio, elaboración, construcción y transición que terminan con un hito. Al final de cada ciclo se tiene una versión del producto.

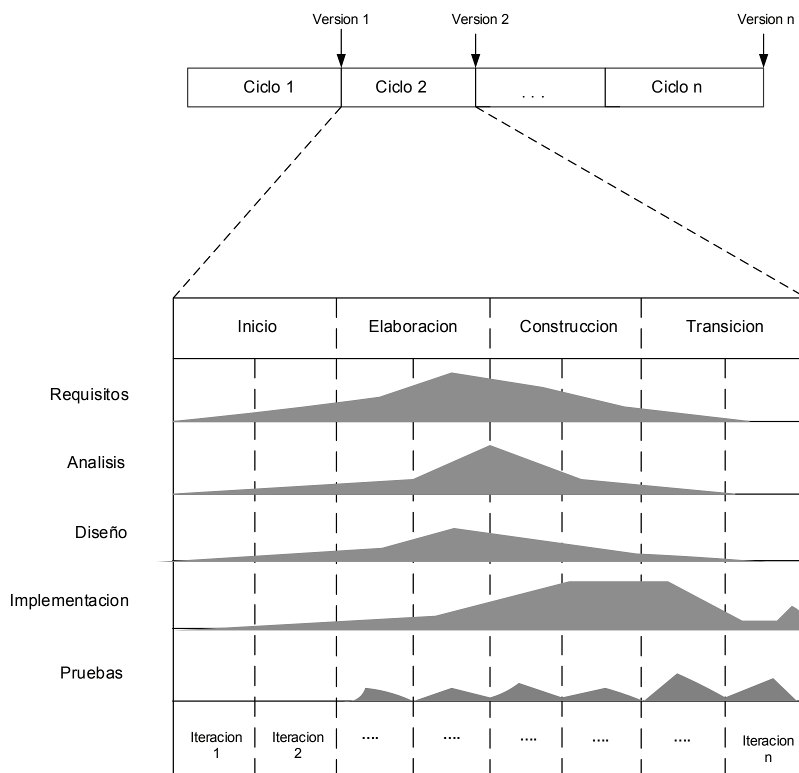

Figura 1.6. Metodología Rational Unified Process (RUP)[Arias07]

Según [Larman02] RUP organiza el trabajo de acuerdo a las cuatro fases anteriormente citadas:

- *Inicio*: Se da una visión aproximada, se realiza un análisis del negocio y se estudia el alcance.
- *Elaboración*: Se realiza una versión más refinada y una implementación iterativa del núcleo central de la arquitectura del sistema. Se identifican riesgos altos, alcance y requisitos.
- *Construcción*: Se implementan iterativamente el resto de los requisitos fáciles y de menor riesgo. Se prepara el despliegue.
- *Transición*: Se liberan pruebas beta.

El producto terminado en RUP debe incluir el código ejecutable, requisitos, casos de uso, especificaciones no funcionales y casos de prueba.

RUP está basado en UML y es soportado por muchas herramientas CASE.

Además de las características enumeradas arriba, RUP construye en componentes conectados entre sí a través de interfaces, que darán lugar a una arquitectura en la cual se fundamenta esta metodología de software (Capítulo 5).

1.5 CALIDAD DEL SOFTWARE

1.5.1 Definición y conceptos

El concepto de calidad siempre ha sido difícil de definir debido a su alto grado de subjetividad. Así, la calidad se convierte en un concepto totalmente relativo: lo que algo es de valor para unas personas puede no serlo para otras. Según el Diccionario de la Real Academia de la Lengua (DRAE) en su 23ª edición, el vocablo *calidad* tiene las siguientes acepciones: 1. f. Propiedad o conjunto de propiedades inherentes a algo, que permiten juzgar su valor. Esta tela es de buena calidad. 2. f. Buena calidad, superioridad o excelencia. La calidad de ese aceite ha conquistado los mercados. 3. f. Adecuación de un producto o servicio a las características especificadas. Control de la calidad de un producto. 4. f. Carácter, genio, índole. 5. f. Condición o requisito que se pone en un contrato. 6. f. Estado de una persona, naturaleza, edad y demás circunstancias y condiciones que se requieren para un cargo o dignidad. 7. f. Nobleza del linaje. 8. f. Importancia o gravedad de algo. 9. f. pl. Prendas personales. 10. f. pl. Condiciones que se ponen en algunos juegos de naipes.

En lo que respecta al domino del software, una definición ampliamente aceptada es la del estándar IEEE 610-1991: "*El grado con el que un sistema, componente o proceso cumple los requisitos especificados y cubre las necesidades o expectativas del cliente o usuario*". Obviamente, el concepto de calidad en el software difiere de cualquier otro producto de la industria, debido a su naturaleza abstracta e intangible, entre otros aspectos.

1.5.2 Dimensiones de la calidad

Aunque existen otras clasificaciones de las dimensiones de la calidad del software, como las de David Garvin [Gar87], aquí se presentarán las de Sebastián, Bargueño y Novo [Minguet03]:

- **Prestaciones**: Se entenderá por prestaciones aquellas características operativas y principales de un producto que son fundamentalmente medibles, como por ejemplo: el tamaño en GB de un una unidad de estado sólido, el número de núcleos de un procesador, etc. La relación entre calidad y prestaciones puede variar dependiendo de las preferencias del usuario. Así, un usuario puede preferir la velocidad de procesamiento a la capacidad de almacenamiento de un equipo y viceversa.
- **Diferenciación**: Son las características secundarias del producto que le proporcionan un valor añadido, por ejemplo, un navegador GPS en un automóvil.
- **Fiabilidad**: Se define como la probabilidad de que un producto no falle en un determinado intervalo de tiempo, por ejemplo, software en tiempo-real tolerante a fallos en un quirófano. Es una medida objetiva.
- **Conformidad**: Se define como el grado en el que el diseño y las prestaciones del producto satisface los estándares establecidos, en general relacionados con la tasa de defectos, número de reclamaciones o reparaciones en período de garantía. También es una medida objetiva.
- **Duración**: Con duración nos referiremos a la estimación de la vida del producto antes de su deterioro sin posibilidad de reparación.
- **Asistencia técnica**: Es uno de los factores que más afecta a la calidad actualmente y se refiere a la atención técnica y de reparaciones de una forma competente y rápida. Algunos aspectos pueden medirse objetivamente.
- **Estética**: Es la dimensión más subjetiva y está supeditada a los sentidos.

1.5.3 Crisis del software y necesidad de una medida (métricas)

A mediados de la década de los sesenta y a medida que los programas se volvían más grandes y más complejos, surgió la necesidad de migrar de las técnicas artesanales a un modo de trabajo disciplinado y ordenado. La magnitud del problema llegó a tal extremo que fue necesario convocar al Comité Científico de la OTAN donde expertos informáticos, basándose en las disciplinas de otras de ingenierías tradicionales, propusieron el concepto de *Ingeniería del Software*. Más tarde, en los años noventa, y a consecuencia de este fenómeno, se reconoció también la necesidad de una medida del software mediante el establecimiento de estimaciones formales, algorítmicas y matemáticas que pudieran, mediante pruebas empíricas, determinar la calidad del software. Así, surgen modelos de medición como COCOMO (COnstructive COst MOdel), SLIM (Software, LIfe Cycle Management) y los conocidos *Puntos Función*. Más tarde se incorporarán métricas para medir la productividad, la complejidad, la fiabilidad, los modelos de datos, la eficacia de los algoritmos criptográficos o la seguridad de la red.

1.5.4 Normalización y certificación

Las metodologías anteriormente explicadas y los modelos de proceso de gestión del software como CMM (Modelo de la Madurez de la Capacidad), la norma ISO 15504 y la metodología METRICA Versión 3 del Ministerio de Administraciones Públicas se consideran estrategias para mejorar la calidad del software [Minguet03].

Además de los conceptos de metodología y modelo, surge el concepto de norma. Dichas normas son establecidas por organizaciones nacionales o internacionales que adoptan los citados modelos y metodologías. En muchas ocasiones las empresas deciden certificar su grado de cumplimiento respecto a una determinada normativa, como por ejemplo las normativas de la ISO (Organización Internacional para la Estandarización). Su homólogo nacional se denomina AENOR (Asociación Española de Normalización) que es el organismo responsable de inspeccionar las empresas y expedir el correspondiente certificado de calidad. En España, la certificación más extendida es la asociada a la norma de ISO 9001:2000 [Minguet03] que especifica los requisitos para un Sistema de Gestión de la Calidad (SGC) en las empresas y asegurar la satisfacción del cliente de sus productos [Cuevas03].

1.5.5 La norma ISO/IEC 9126

La norma ISO/IEC 9126 nace como norma ISO de medida de calidad del software interna y externa en forma de descomposición jerárquica en árbol. Está basada en los modelos de calidad de McCall y Bohem, autores que sentaron un

precedente en el campo de la calidad del software. A continuación se expone la tabla sinóptica de características y subcaracterísticas de la norma[5]:

Calidad del software (interna y externa)					
Funcionalidad	**Fiabilidad**	**Facilidad de uso**	**Eficiencia**	**Mantenimiento**	**Movilidad**
Idoneidad	Madurez	Fácil comprensión	Comportamiento frente al tiempo	Facilidad de análisis	Adaptabilidad
Exactitud	Tolerancia a fallos	Fácil aprendizaje	Uso de recursos	Capacidad para cambios	Facilidad de instalación
Interoperatividad	Capacidad de recuperación	Operatividad	Adherencia a normas	Estabilidad	Coexistencia
Seguridad	Adherencia a normas	Software atractivo		Facilidad para pruebas	Facilidad de reemplazo
Adherencia a normas		Adherencia a normas		Adherencia a normas	Adherencia a normas

Figura 1.7. La norma ISO/IEC 9126

1.5.6 Aseguramiento de la calidad del software (SQA)

El software se encuentra cada vez más presente en nuestras vidas y en muchos elementos de nuestro entorno. Por este motivo, la implicación en el riesgo del software propenso a errores es una cuestión estrechamente ligada a la calidad. Según [Cuevas03], "El objeto del SQA es proporcionar a la dirección una perspectiva adecuada del proceso que utiliza el proyecto software y de los productos a crear. El Aseguramiento de la Calidad Software implica la revisión y la inspección de los productos y las actividades del software, con objeto de verificar que se cumplen los procedimientos y estándares aplicables, y de proporcionar a los responsables del proyecto software y a otros directivos los resultados de dichas revisiones e inspecciones".

Por tanto, la misión del SQA es:

- Planificar las actividades de aseguramiento de la calidad.
- Verificar que los productos y el proceso software cumplen los requisitos, estándares y procedimientos establecidos.

---

5 Para más información consultar [Minguet03].

- Informar a la dirección y a las personas relacionadas con el proyecto sobre los resultados del aseguramiento de la calidad.
- Elevar a la dirección los aspectos que no pueden solucionarse en el contexto del proyecto.

Las actividades que recaen sobre el grupo SQA son:

- Realizar el *plan*[6] SQA en las primeras fases del proyecto.
- Establecer las actividades del personal SQA respecto al plan SQA.
- Participar en la definición y revisión del plan del proyecto. El grupo SQA asesora en cuanto elaboración y revisión de planes, normas y procedimientos de desarrollo software.
- Revisar las actividades de desarrollo software con el fin de verificar su cumplimiento.
- Inspeccionar los productos software con el propósito de verificar su corrección (antes de entregar el producto se evalúa en función a estándares, procedimientos y requisitos del contrato).
- Informar periódicamente al equipo de desarrollo.
- Documentar las desviaciones de las actividades y productos software.
- Revisar periódicamente las propias actividades del grupo SQA con el personal SQA del cliente.

Las actividades de gestión, aseguramiento y control de la calidad se realizan mediante técnicas para detectar defectos en el software principalmente. Las técnicas se dividen en *estáticas* o *dinámicas.* Las técnicas estáticas se realizan sin ejecutar el software, como las auditorías; mientras que las técnicas dinámicas se realizan mediante casos de prueba que suponen la ejecución del software real y el análisis manual o automatizado del código fuente. Estos casos de prueba pueden ser de *caja blanca* (cobertura de sentencias, decisiones, ramificaciones, condiciones y caminos) y *caja negra* (particiones de equivalencia, análisis de valores límite, conjetura de errores, gráficas de causa-efecto, etc.).

[Cuevas03].

6 Para más información del plan SQA consultar [Cuevas03] y el estándar IEEE 730-1989 allí descrito.

1.6 DESARROLLO DE SOFTWARE SEGURO

1.6.1 Introducción

La utilización de software en entornos críticos como el militar, transportes, gubernamental o médicos implica la protección de datos personales tales como bases de datos, ficheros, aplicaciones, memoria del sistema y unidades de procesamiento. Por ello, uno de los aspectos del SQA es la de velar por la seguridad del software con el objetivo de satisfacer factores de calidad como integridad, disponibilidad y fiabilidad.

Es ampliamente asumida la importancia de la seguridad en sistemas computacionales en lo concerniente a redes sociales, aplicaciones móviles, la nube o en software ubicuo. Los ataques a dichos sistemas aprovechan las vulnerabilidades del software para conseguir acceso a través de técnicas como el *phishing*, el *malware* y el espionaje de datos por *hackers* o entidades no autorizadas. Por este motivo es necesario la construcción de software confiable que garantice al usuario su utilización segura. Para ello es imprescindible un diseño con una arquitectura robusta que prevenga, repela y se recupere de los ataques maliciosos. Puesto que las consecuencias de un fallo o un ataque pueden concernir vidas humanas o costes millonarios, es de suma importancia anticipar e identificar las condiciones o amenazas que puedan causar daños o vulnerabilidades. A este proceso se le denomina *análisis de amenazas*.

Las actividades recomendadas para la construcción de software seguro recomiendan no aplicar métodos *ad hoc* o sobre la marcha para ir parcheando errores y fallas de seguridad, pues es ineficiente y muy costoso.

Una de las recomendaciones es realizar revisiones de código antes de las pruebas para evitar fallos potenciales y mejorar la calidad del software. Durante la planificación del proyecto debe tenerse en cuenta el presupuesto y temporización de los objetivos de seguridad. En el análisis de riesgos se debe estimar el coste asociado a la perdida de datos o recursos producidos por fallos de software o ataques malintencionados. Así mismo, la identificación de amenazas al sistema a menudo se retrasa hasta que los requisitos de un incremento software son trasladados a sus requisitos de diseño. Las revisiones de código centradas en aspectos de seguridad deben incluirse en la implementación y deben basarse en los objetivos y amenazas de seguridad identificados en las actividades de diseño.

Respecto a la fase de pruebas, el aseguramiento de la seguridad debe demostrar que se ha construido un sistema seguro que inspira confianza. Por este motivo, una de las actividades del aseguramiento de la seguridad es la realización de verificaciones que proporcionen evidencias de que el software cumple los requisitos.

El aseguramiento de la seguridad se compone de una serie de artefactos auditables, llamados *casos de aseguramiento*, que confirman que el software satisface las demandas de seguridad. Como casos de aseguramiento pueden utilizarse, aunque no es suficiente, las pruebas formales de software y herramientas de análisis de vulnerabilidades como RATS (*Rough Auditing Tool for Security),* ITS4 para C++ y SLAM desarrollado por Microsoft. [Pressman14].

1.6.2 Análisis de software seguro

1.6.2.1 ANÁLISIS DE REQUISITOS

Es importante saber que los requisitos de seguridad son requerimientos no funcionales que influyen fuertemente en la posterior arquitectura del software. Una vez que se los requisitos de seguridad han sido analizados mediante el modelo de amenazas y el análisis de riesgos, se procede a crear un conjunto de políticas de seguridad. Una política de seguridad proporciona una definición de seguridad que incluye requisitos clave de seguridad y una serie de reglas que describen cómo aplicar la seguridad durante el funcionamiento del software. Puede que los requisitos de seguridad entren en conflicto con otros requisitos convencionales del software, tales como seguridad y usabilidad. No es algo infrecuente. Según [Marick02] el análisis de riesgos tiene que considerar los siguientes preceptos:

- Las necesidades de los usuarios respecto a la seguridad.
- La seguridad del diseño arquitectónico para que se ajuste a un buen diseño de interfaz de usuario.
- Una interfaz de usuario segura pero que al mismo tiempo posibilite aspectos como efectividad y eficiencia.

Durante el análisis de requisitos, el analista puede utilizar *patrones de ataque* que facilitan en gran medida la reutilización de escenarios de vulnerabilidad al modo problema → solución, como por ejemplo *phishing, SQL injection*, etc.

1.6.2.2 MODELADO DE SEGURIDAD

Un modelo de seguridad es una descripción formal de una política de seguridad y puede ser una valiosa guía durante el diseño, implementación y el proceso de revisión. Un modelo de seguridad suele representarse de forma textual o mediante formalismos gráficos como las máquinas de estado (Capítulo 11). El ingeniero de software debe asegurarse que las transiciones de la máquina de estados finita comiencen en un estado seguro y que finalicen en otro estado seguro. Otros

formalismos gráficos utilizados en el modelado se seguridad están basados en el lenguaje **UMLsec** que comprende una extensión de UML utilizando estereotipos y restricciones y GRL que es otro lenguaje que permite la captura de requisitos no funcionales del sistema.

1.6.2.3 MÉTRICAS DE SEGURIDAD

Un software seguro debe cumplir los siguientes tres preceptos[7]:

1. Operar en situaciones hostiles.
2. El software no se comportará de una forma maliciosa.
3. Funcionar en cualquier situación comprometida.

Las citadas tres propiedades deben ser tenidas en cuenta por cualquier métrica de seguridad. Dichas métricas deben proporcionar a los desarrolladores una medida del nivel de confidencialidad de los datos y la integridad del sistema que podría estar en riesgo.

1.6.2.4 COMPROBACIONES DE SEGURIDAD

Las comprobaciones de seguridad deben realizarse durante todo el ciclo de vida del software utilizando auditorías, inspecciones y casos de prueba, pero fundamentalmente al comienzo del proceso de desarrollo. En las actividades del SQA se incluyen los estándares de seguridad y guías de desarrollo seguro para ser utilizados durante el modelado y la construcción del software. Las actividades de verificación comprueban que los casos de prueba de seguridad se han realizado correctamente y pueden ser trazados hasta los requisitos del sistema. Los datos recogidos durante las comprobaciones son analizados en los casos de aseguramiento descritos en la sección 1.6.1.

[Pressman14].

1.6.3 Análisis de riesgos

El análisis de riesgos es una actividad crítica en la planificación del proyecto. El método del *modelado de amenazas* se utiliza en el análisis de riesgos con el propósito de identificar posibles daños al software y debe realizarse en las primeras fases del ciclo de vida utilizando los requisitos y los modelos de análisis. *El modelo*

---

7 *https://www.us-cert.gov/bsi*

de amenazas debe identificar los componentes clave del sistema, descomponerlo, clasificar las amenazas de cada componente del software por nivel de riesgo y desarrollar estrategias de migración. [Pressman14].

1.7 MDA

En el año 2000 el *Object Management Group* (OMG) publicó su informe sobre la *Arquitectura Dirigida por Modelos* MDA **(***Model-Driven Architecture***),** un paradigma de construcción de aplicaciones que innovaría por completo el panorama actual de las herramientas CASE de última generación para el desarrollo de aplicaciones. La idea principal en la que se basa MDA es motivar a desarrollar principalmente con modelos, con el fin de independizarse de la plataforma y el lenguaje de implementación. Muy recientemente ha surgido el paradigma MDE **(***Model-Driven Engineering***)** que es más avanzado y se basa en modelos abstractos del dominio más que en algorítmica computacional (léase [Brambilla17]).

1.7.1 Introducción

Desde siempre, la construcción de software complejo se ha vuelto económicamente muy costosa. La demanda de software continúa en auge y en los últimos años ha habido un incremento muy significativo en la industria del software. Las técnicas tradicionales de programación se han quedado obsoletas para abordar la complejidad y el coste del desarrollo de los futuros sistemas informáticos. Esto ha suscitado la aparición de enfoques, como el MDA, orientados a aumentar el nivel de abstracción en el desarrollo.

En los años 50 el desarrollo se realizaba únicamente introduciendo los códigos binarios directamente a la máquina, posteriormente, en los años 60 se inventó el lenguaje ensamblador que permitiría asociar códigos nemotécnicos a un conjunto de bits que representaban una instrucción. Con la llegada de los primeros lenguajes de alto nivel (Algol, Fortran, C), se pudo dar un paso más en el aumento del nivel de abstracción, convirtiendo sentencias sintácticas complejas en instrucciones en ensamblador. Finalmente, en los 80 emergió la Programación Orientada a Objetos que agilizó enormemente la productividad del software. El último paso en la evolución del desarrollo de software ha sido convertir los modelos textuales o gráficos en código ejecutable gracias a compiladores basados en MDA.

Además del nivel de abstracción, otro de los objetivos que persigue MDA es la posibilidad de aumentar el nivel de reutilización, permitiendo la construcción de software con bloques de diseño reutilizables. La otra gran ventaja que permite esta tecnología es la *Interoperabilidad de Tiempo-Diseño***,** o lo que es lo mismo, la

facilidad de desarrollar una aplicación independiente de su implementación lo que permite recombinar tecnologías.

Aunque la reutilización ha conseguido un gran avance en el mundo de la computación, aún existe un problema: hay poca reutilización de aplicaciones. Las librerías y los *frameworks* han conseguido un desarrollo espectacular, aun así se los considera muy cercanos a nivel de la máquina.

1.7.2 Características de MDA

Modelos: Con los modelos permitimos elaborar una descripción abstracta del mundo. MDA permite modelar la aplicación con entidades que representan objetos del mundo concreto o abstracto; instrumentos como la abstracción y la clasificación permiten jerarquizar y ordenar los conceptos componentes del sistema. Para conseguir todas estas cosas, MDA cuenta con el uso de UML que es el estándar oficial para el modelado de software; aunque actualmente se están imponiendo los DSL **(***Domain Specific Languages* o Lenguajes específicos del dominio) que ya son una alternativa a UML y pueden ser textuales o gráficos.

Metamodelos: Un metamodelo es simplemente un modelo de un lenguaje de modelado como UML. Así, por ejemplo, un metamodelo para UML describe cómo éste crea las clases, sus relaciones y jerarquías en otro modelo más completo que el anterior y que recoge detalladamente cada característica del modelo que describe.

El metamodelo de UML se expresa mediante MOF **(***Meta Object Facility***),** una herramienta estandarizada por la OMG. Un ejemplo es XMI, que es el acrónimo de (XML Metadata Interchange) y permite definir cómo se describen e intercambian los modelos. Muchas de las herramientas CASE actuales como Rational Rose, MagicDraw o StarUML utilizan este formato XML para expresar sus metamodelos.

Transformaciones entre modelos: La idea que se persigue con la transformación de modelos es poder convertir un modelo en otro similar, ya sea por refinamiento o abstracción. En este sentido sería posible transformar lo que se denomina PIM *(Platform Indepenent Model)* en un PSM *(Platform Specific Model).* Para ello se utilizan las denominadas reglas de transformación y las marcas que permiten asociar y ajustar el nivel de detalle en dichas transformaciones. Se puede concebir las marcas como una especie de filtro, en el que por un lado se introduce la transformación a realizar y por otro lado obtenemos la transformación realizada. El objetivo de estos es transformar un modelo de entrada en otro modelo de salida en el que, por ejemplo, se haya realizado una transformación para obtener un mayor nivel de refinamiento. Piense, por ejemplo, en una transformación entre un DSL textual para el lenguaje ensamblador al código específico de una determinada arquitectura hardware.

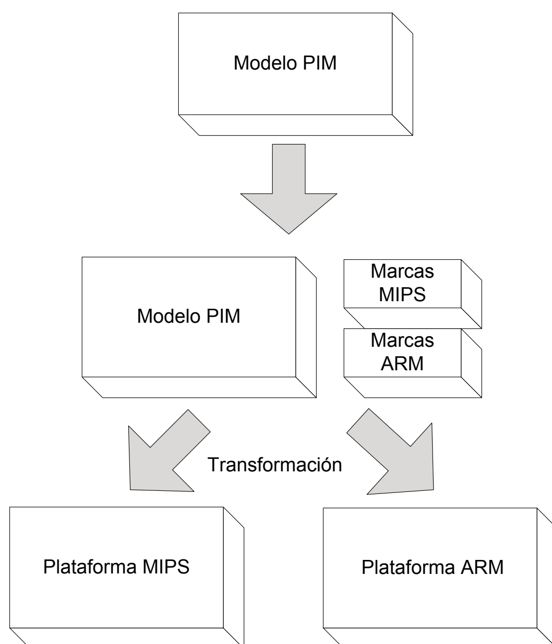

Figura 1.8. Transformación del PIM al PSM

Utilización de lenguajes: Una de las características más interesantes de MDA es la posibilidad de extender las capacidades de UML por medio de los perfiles (profiles). Estos se adaptan "*ad-hoc"* a cualquier dominio, plataforma o método que se elija, proporcionando una versatilidad y una potencia que alcanza una gran variedad de lenguajes y tecnologías. De esta forma es posible encontrar herramientas MDA en el mercado que ofrecen perfiles tan variados como EJB, C++, Ruby, CORBA, etc. Una de las maneras de ampliar la potencialidad de los lenguajes ofrecidos por una herramienta de estas características es por medio de los *estereotipos***,** los cuales permiten extender el vocabulario de UML, así como las *restricciones* (constraints) que especifican condiciones que deben cumplirse dentro de un modelo para su correcto funcionamiento.

Modelos ejecutables y MDA ágil: La idea subyacente de los modelos ejecutables es que se comporten directamente como código, facilitando la interacción con el dominio del cliente. La finalidad de este objetivo MDA es la independencia de la plataforma consiguiendo, entre otros aspectos, que tanto los modelos como el código sean una única entidad y que puedan ser ejecutados tan pronto como finalice el diseño de los mismos, proporcionando una realimentación con los clientes y expertos del dominio.

Para entender estas ideas supongamos que tuviera que desarrollar una aplicación con tres posibles sistemas operativos, tres bases de datos y tres implementaciones de *Servicios Web* (SOA). Se tendría en total $3 \times 3 \times 3 = 27$ implementaciones diferentes, lo que se concluye que la reutilización a nivel de código es multiplicativa, no aditiva.

Aplicación
Base de Datos (SGBD)
SOA
Plataforma (Sistema Operativo + Hardware)

Figura 1.9. Problema de la reutilización multiplicativa

En la figura 1.9 podemos observar el problema anteriormente citado. Para conseguir una solución aditiva que permita reutilizar una capa independientemente de las otras se deben interconectar con mecanismos que sean independientes de los contenidos de las otras capas. Esto supone un gran avance, pues MDA impone la arquitectura únicamente en el último momento. En general las ventajas que se consiguen son muchas, entre ella la posibilidad de no tener que modificar los modelos para generar el código final en caso de cambiar alguna capa, el coste bajo del desarrollo multiplataforma, una productividad y un mantenimiento barato.