El proceso de desarrollo de *software*

[2.1] ¿Cómo estudiar este tema?

[2.2] Definición

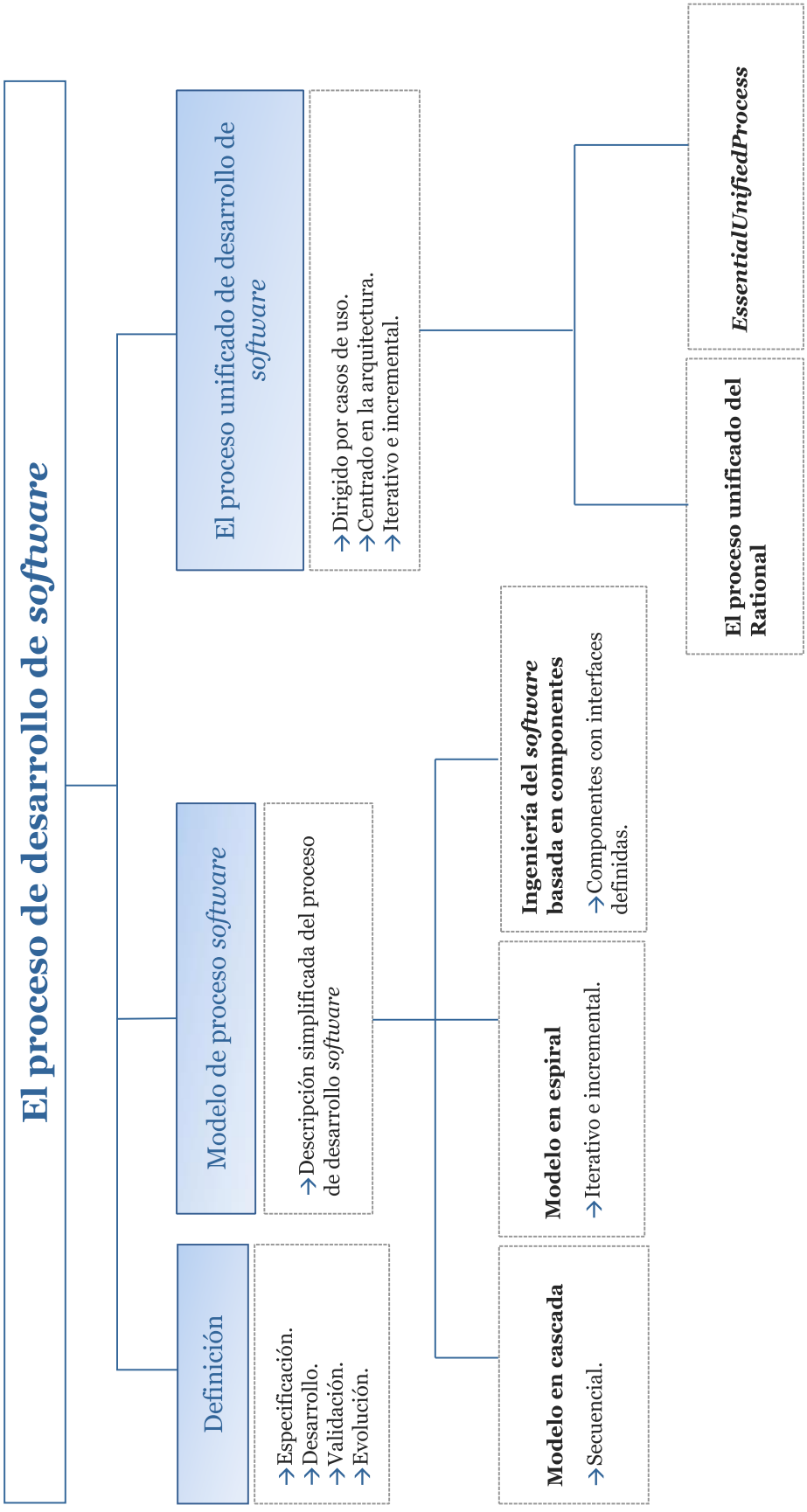
[2.3] Modelo de proceso software

[2.4] El proceso unificado de desarrollo de *software*

[2.5] Referencias bibliográficas

TEMA 2

Esquema


El proceso de desarrollo de *software*
Definición
→Especificación.
→Desarrollo.
→Validación.
→Evolución.
Modelo de proceso *software*
→Descripción simplificada del proceso de desarrollo *software*
Modelo en cascada
→Secuencial.
Modelo en espiral
→Iterativo e incremental.
Ingeniería del *software* basada en componentes
→Componentes con interfaces definidas.
El proceso unificado de desarrollo de *software*
→Dirigido por casos de uso.
→Centrado en la arquitectura.
→Iterativo e incremental.
El proceso unificado del Rational
EssentialUnifiedProcess

Ideas clave

2.1. ¿Cómo estudiar este tema?

Para estudiar este tema **lee las «Ideas clave»**, además del capítulo 4 (páginas 59-78), del libro:
Sommerville, I. (2005). *Ingeniería del Software*. Séptima edición. España: Pearson Addison-Wesley.

En este tema se estudia el significado de proceso de desarrollo de *software*, así como los modelos de proceso más comunes que se suelen aplicar a la hora de implementar una aplicación. Las características propias del proyecto *software* harán que, según el caso, se deba utilizar un modelo u otro de proceso. Además, se analiza el proceso unificado de desarrollo de software como ejemplo de modelo de proceso que pretende dar solución a los problemas que surgen cuando se desarrolla un sistema *software*, es decir, pretende facilitar la gestión de un proyecto *software* complejo, en lo que respecta a la coordinación de las distintas tareas que se deben realizar a lo largo de todo el proceso de desarrollo.

2.2. Definición

Un proceso, de manera genérica, define «quién está haciendo qué, cuándo, y cómo alcanzar un determinado objetivo» (Jacobson, Booch y Rumbaugh, 2000, prefacio). Para **la ingeniería de *software*** dicho **objetivo sería el desarrollo de un producto *software*, o bien, la modificación o ampliación de uno ya existente**. La definición y aplicación de un proceso de desarrollo cuando se inicia un proyecto *software* se debe a ese intento de mejorar la forma de trabajar (más disciplinada y sistemática), favoreciendo la prevención y resolución de posibles problemas que puedan surgir en dicho proceso (Humphrey, 1990, 255).

En la literatura se pueden encontrar varias definiciones para el proceso de desarrollo de *software*. Jacobson, Booch y Rumbaugh (2000, 4) definen el proceso de desarrollo de *software* como «el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema *software*». Otra definición similar para el *proceso de desarrollo de software* es la proporcionada por Sommerville (2005, 60) que lo define como «un conjunto de actividades que conducen a la creación de un producto *software*».

En la misma línea, otra definición más elaborada se encuentra en (Pressman, 2010, 27) que describe el proceso de desarrollo como un conjunto de actividades estructurales, donde cada una de estas actividades estará compuesta por un conjunto de acciones de ingeniería de *software*, que a su vez vendrán definidas por un conjunto de tareas que identificarán el trabajo a realizar, los productos que se generan en cada trabajo, y los puntos de aseguramiento de la calidad que son requeridos, así como los puntos de referencia que se van a utilizar para evaluar el avance del producto *software*.

Por último, bajo otra perspectiva, Humphrey (1990, 3) define el proceso de desarrollo de *software* como «el conjunto de herramientas, métodos y prácticas que utilizamos para crear un producto software». Independientemente del enfoque de la definición, los ingenieros *software* serán los responsables de llevar a cabo gran parte de ese conjunto de actividades y de seleccionar las herramientas, métodos y prácticas más adecuados para desarrollar el *software* requerido.

La ejecución de cualquier proceso de desarrollo de *software* comprenderá un conjunto de actividades que van a generar artefactos de dos tipos diferentes:

- Artefactos internos del proceso de desarrollo.
- Artefactos entregables al cliente.

Un **artefacto se puede definir** como toda pieza de información que se utiliza o se produce en un proceso de desarrollo de *software*.

El motivo por el que se siguen definiendo procesos de desarrollo de *software* se debe a un intento de mejorar la forma de trabajar cuando se construye *software*. De este modo, si se piensa en el proceso de desarrollo de una manera ordenada, resultará más sencillo anticiparse a ciertos problemas y elaborar formas de prevenir y resolver riesgos potenciales. Tal y como se destaca en (Humphrey, 1990), algunos de los aspectos de mayor interés en los que se centra el proceso *software* tienen que ver con la calidad, la tecnología del producto, la inestabilidad de los requisitos, y la complejidad de los sistemas.

Todo proceso de desarrollo de *software* debe definir el problema para favorecer su comprensión y diseñar su solución. Para ello, tal y como se muestra en la Figura 1 se partirá de un conjunto de requisitos que se ha de ser capaz de transformar en un producto *software* que los satisfaga.

Figura 1. Transformación de los requisitos de usuario en un producto *software*.

Según Sommerville (2005, 7), existen cuatro actividades fundamentales que son comunes para todos los procesos de desarrollo de *software*:

- **Especificación del *software***. Clientes e ingenieros definen el *software* a construir junto con sus restricciones.
- **Desarrollo del *software***. El *software* se diseña y se implementa, a la vez que se verifica que se construye de manera correcta.
- **Validación del *software***. El *software* se valida, para comprobar que el *software* es correcto, es decir, hace lo que el cliente había solicitado. La validación se llevará a cabo teniendo en cuenta los requisitos de usuario.
- **Evolución del *software***. El *software* se modifica para adaptarse a los cambios requeridos por el cliente o por el mercado.

Cada actividad comprenderá a su vez de un conjunto de tareas que deberán ser realizadas por recursos concretos, no siempre necesariamente humanos. Estas tareas se podrán organizar de distinta manera en el tiempo, y con diferente nivel de detalle según las características y la complejidad del *software* a desarrollar.

2.3. Modelo de proceso *software*

Un **modelo de proceso de desarrollo de *software*** es una **descripción simplificada** de un proceso de desarrollo de *software* real. Aunque se trata de una representación simplificada del proceso *software*, precisamente esta característica constituye una de sus principales ventajas: un modelo de proceso *software* debería ser fácil de entender y de seguir por todos los desarrolladores que participan en un proyecto determinado.

Ante un proyecto *software*, las organizaciones adoptarán el modelo de proceso *software* que mejor se adapte, lo que dependerá de la complejidad y las características del sistema

a construir. Hay que tener en cuenta que la elección inadecuada de un modelo de proceso puede tener un gran impacto en la calidad o utilidad del sistema *software* construido.

Algunos ejemplos típicos de modelos de proceso que se pueden adoptar a la hora de construir *software* se describen en los siguientes subapartados.

Modelo en cascada

El modelo en cascada (*waterfall model*) fue el primer paradigma de proceso de desarrollo de *software* reconocido (ver Figura 2), y se deriva de otros procesos de ingeniería utilizados para la construcción de productos físicos. Este modelo toma las actividades comunes para todos los procesos de desarrollo de *software* (especificación, desarrollo, validación y evolución), y las representa como fases individuales secuenciales: especificación de requisitos, diseño de *software*, implementación, pruebas, etc. El resultado de cada fase ha de ser aprobado (*signed off*). Una fase no puede comenzar hasta no haber finalizado (aprobado) la anterior. Desgraciadamente, este modelo de proceso resulta poco realista, ya que difícilmente se van a encontrar proyectos *software* reales que se ajusten a este tipo de modelo de proceso de manera tan rigurosa.

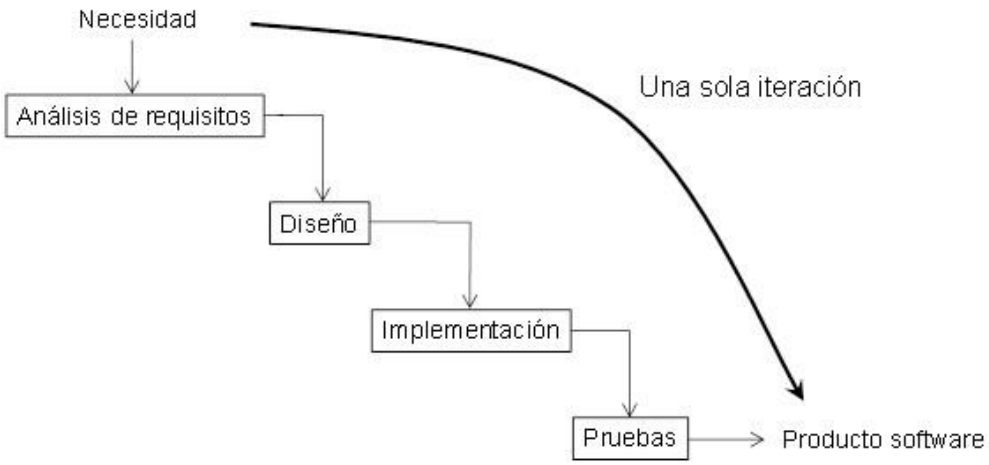


Figura 2. Ciclo de vida del desarrollo de software en cascada

Modelo en espiral

El modelo en espiral constituye una técnica propuesta por Boehm (1988) y representa el proceso *software* como una espiral (ver Figura 3), que comprende a la vez el modelo iterativo y el modelo incremental (ver Figura 4).
De este modo, este modelo de proceso está diseñado explícitamente para soportar iteraciones en el proceso, donde en cada iteración se va añadiendo más funcionalidad al sistema (es decir, un incremento) hasta llegar a una iteración que contiene el sistema completo.

Además, en cada bucle de la espiral se representa una fase del proceso *software*. El bucle más interno trata la viabilidad del sistema, el siguiente bucle la definición de los requisitos del sistema, el siguiente bucle el diseño del sistema, y así sucesivamente.

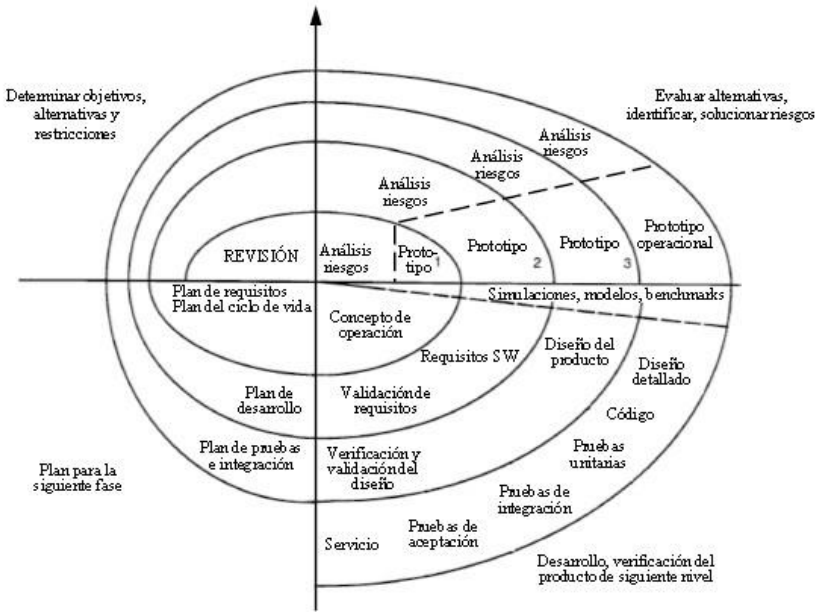


Figura 3. Modelo en espiral (iterativo + incremental)

Por otro lado, cada bucle de la espiral se divide en cuatro sectores: (i) **establecimiento de los objetivos**, donde se definen los objetivos específicos para cada fase del proyecto; (ii) **evaluación y reducción de riesgos**, donde se lleva a cabo un análisis detallado de cada uno de los riesgos del proyecto que se hayan identificado; (iii) **desarrollo y validación**, donde una vez que se ha llevado a cabo la evaluación de riesgos, se selecciona una técnica de desarrollo para el sistema (por ejemplo, el modelo en cascada podría ser el desarrollo más apropiado si el riesgo identificado más relevante es la integración de subsistemas); y (iv) **planificación**, donde se revisa el proyecto y se toma la decisión de si se ha de continuar con un nuevo bucle de la espiral. Si se decide continuar, entonces se planifica la siguiente fase del proyecto. La distinción más

importante entre el desarrollo en espiral y otras técnicas de procesos de *software* es la consideración explícita de riesgos en el proceso en espiral.

(a) Enfoque iterativo

(b) Enfoque incremental

Figura 4. Iterativo vs Incremental (Fuente: http://jan-so.blogspot.com.es/2008/01/difference-betweeniterative-and.html)

Ingeniería de *software* basada en componentes

Aunque sea de manera informal, *ad hoc*, en la mayoría de los proyectos *software* siempre se va a dar algo de reutilización, ya que es muy raro que las organizaciones construyan sistemas completamente nuevos, es decir, que partan de cero (Sommerville, 2005, 6466). Bajo esta perspectiva, **la técnica de ingeniería de *software* basada en componentes** (del inglés *Component-Based Software Engineering*, CBSE), orientada a la reutilización, se basa en la existencia de un número significativo de componentes reutilizables.

En este contexto, un componente se define como «un elemento *software* que se adecúa a un modelo de componentes y que puede ser distribuido de manera independiente e integrado sin ningún tipo de modificación siguiendo un estándar de composición» (Heineman y Council, 2001). Para CBSE, el proceso de desarrollo del sistema se centra, por tanto, en integrar estos componentes de manera sistemática en vez de desarrollar el sistema partiendo de cero. El modelo de proceso genérico para CBSE se muestra en la Figura 5.

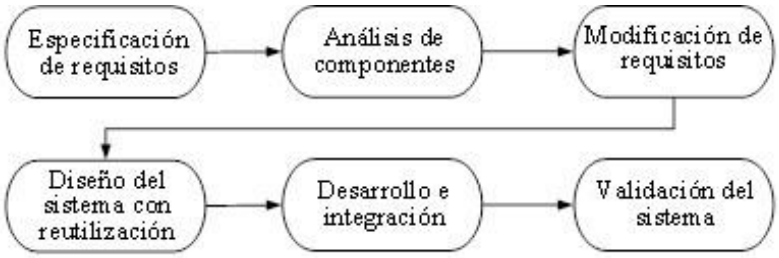


Figura 5. Ingeniería de *software* basada en componentes. Fuente: Sommerville (2005).

A este nivel, merece la pena mencionar un enfoque muy relacionado con CBSE, aunque no son lo mismo, conocido como *Arquitectura Orientada a Servicios* (del inglés *Service Oriented Architecture*, SOA). En este caso, la funcionalidad del producto *software* se proporciona en base a un conjunto de servicios *software* que se comunican a través de mensajes que pueden ser invocados de manera remota, y que son conformes a un conjunto definido de protocolos y formatos de datos.

Desde un punto de vista conceptual, se podría considerar un servicio como un componente de una solución global, aunque en realidad es el servicio el que internamente está formado por componentes *software* que se comunican entre sí a través de interfaces bien definidas (Microsoft, 2002). Las principales características del modelo de SOA son las que se indican a continuación (Microsoft, 2009):

- **Autónomo.** Cada servicio se mantiene, se desarrolla, se despliega y evoluciona de manera independiente.
- **Distribuido.** Los servicios se pueden localizar en cualquier ubicación de una red, de manera local o remota, siempre y cuando la red soporte los protocolos de comunicación requeridos en cada caso concreto.
- **Débilmente acoplado.** Cada servicio será independiente de los demás, por lo que, si la interfaz sigue siendo compatible, un servicio podrá ser reemplazado o actualizado sin que afecte a la solución global que utiliza dicho servicio.
- **Compatibilidad basada en políticas.** La política, en este caso, hace referencia a la definición de características, tales como transporte, protocolo y seguridad.

Un ejemplo muy típico de aplicación de SOA es el modelo de servicio denominado *Software as a Service* (SaaS) donde los usuarios tienen disponible una cartera (*portfolio*) de aplicaciones que podrían ser útiles para su negocio, y cuyo acceso se realiza a través de un navegador web a la nube (del inglés *cloud computing*), desde donde podrán bajarse la aplicación. Los usuarios podrán configurar esa aplicación como necesiten, pero no van poder gestionar la infraestructura subyacente que controla esa aplicación en la web.

Ejemplos de SaaS serían, entre otros, Google Docs, DropBox, SalesForce, Gmail y Microsoft Office 365.

2.4. El proceso unificado de desarrollo de *software*

El **proceso unificado de desarrollo de software** (del inglés *Unified Software Development Process*, USDP), más conocido de manera simplificada como **proceso unificado** (del inglés *Unified Process*, UP), se corresponde con un proceso de desarrollo de *software* elaborado por los autores del *Lenguaje Unificado de Modelado* (del inglés *Unified Modeling Language*, UML): Grady Booch, James Rumbaugh e Ivar Jacobson.

De acuerdo a sus creadores (Jacobson, Booch y Rumbaugh, 2000), UP es un proceso basado en componentes, lo cual quiere decir que el sistema *software* en construcción está formado por componentes *software* interconectados a través de interfaces bien definidas (es decir, el conjunto de operaciones que son utilizadas para especificar la funcionalidad que ofrece un componente). En este sentido, un componente *software* sería la parte física y reemplazable de un sistema *software* que se ajusta a, y proporciona la realización de, un conjunto de interfaces (Booch, Rumbaugh y Jacobson, 2005, 453).

Además, el proceso unificado utiliza UML para modelar todos los artefactos del sistema *software*. Pero los auténticos aspectos definitorios y distintivos de UP se resumen en los siguientes conceptos básicos (Jacobson, Booch y Rumbaugh, 2000):

- **Dirigido por casos de uso.** El proceso de desarrollo avanza a través de una serie de actividades que parten de los casos de uso. Los casos de uso se especifican, se diseñan y son la fuente a partir de la cual los ingenieros de pruebas construyen los casos de prueba del sistema *software*.
- **Centrado en la arquitectura**. El papel de la arquitectura *software* es parecido al papel que juega la arquitectura en la construcción de edificios. El edificio se contempla desde varias perspectivas: estructura, servicios, conducción de la calefacción, fontanería, electricidad, etc. Esto permite al constructor ver una imagen completa antes de que comience la construcción. Análogamente, la arquitectura en un sistema *software* se describirá mediante diferentes vistas del sistema en construcción (estática, dinámica…). Con lo cual, el concepto de arquitectura *software* incluye los aspectos estáticos y dinámicos más significativos del sistema *software*.
- **Iterativo e incremental.** La idea de este modelo de proceso es dividir el trabajo en partes más pequeñas o mini-proyectos, donde cada mini-proyecto es una iteración

que resulta en un incremento. Las iteraciones hacen referencia a pasos en la actividad, y los incrementos hacen referencia al crecimiento del producto. Al ser mini-proyectos, comienzan con los casos de uso y continúan con las tareas de análisis, diseño, implementación y pruebas, que terminan convirtiendo en código ejecutable los casos de uso que iniciaban la iteración, es decir, cada iteración va a contener todos los elementos de un proyecto de desarrollo de *software*: planificación, análisis y diseño, construcción, integración y pruebas, y una versión interna o externa. Cada iteración genera una línea base (*baseline*) que comprende una versión parcialmente completa del sistema final y toda la documentación del proyecto asociada (para generar una línea base es necesario que todos los artefactos de la iteración que representa hayan sido revisados y aprobados a través de procedimientos formales). La diferencia entre dos líneas base consecutivas es lo que se denominado incremento. Además, en cada iteración, el proceso unificado engloba cinco **flujos de trabajo** principales: (i) **Requisitos**, que recoge lo que el sistema debería hacer; (ii) **Análisis**, que refina y determina la estructura de los requisitos del sistema; (iii) **Diseño**, que realiza los requisitos que se dan en la arquitectura del sistema; (iv) **Implementación**, que construye el *software*; y (v) **Pruebas**, que verifica que la implementación funciona correctamente y cumple los requisitos.

El proceso unificado se repite a lo largo de una serie de ciclos que constituyen la vida de un sistema *software*. Cada ciclo constituye una versión del sistema, y se encuentra dividido en cuatro fases que se describen a continuación (Jacobson, Booch y Rumbaugh, 2000):

- **Inicio.** Durante la fase de inicio se desarrolla una descripción del producto final y se presenta el análisis estratégico para el producto. De esta manera, si la contribución del sistema *software* para el negocio es adecuada, entonces se sigue adelante con el proyecto. En esta fase se genera el modelo de casos de uso con los casos de uso más críticos, se esboza la arquitectura a través de los subsistemas más importantes, se identifican y priorizan los riesgos más importantes, se planifica en detalle la fase de elaboración y se realiza una estimación orientativa del proyecto.
- **Elaboración.** Durante la fase de elaboración se especifican en detalle la mayoría de los casos de uso del producto (es decir, comprensión del dominio del problema) y se diseña la arquitectura del sistema.
- **Construcción.** Durante la fase de construcción se crea el producto. En esta fase la línea base de la arquitectura crece hasta convertirse en el sistema completo. Como ya se comentaba anteriormente, la línea base constituye un conjunto de artefactos revisados y aprobados, que representa un punto de acuerdo para la posterior

evolución y desarrollo del sistema, y que solamente puede ser modificado a través de procedimientos formales.

- **Transición.** Durante la fase de transición se cubre el periodo durante el cual el producto se convierte en versión que se podría llamar *beta*, que se irá refinando hasta su versión final. En esta fase los desarrolladores corrigen los problemas e incorporan las mejoras que se estimen convenientes.

Cada fase termina con un hito (*milestone*) que se determina por la disponibilidad de un conjunto de artefactos (es decir, información que ha sido desarrollada para construir el sistema, y se encuentra en un determinado estado). El número de iteraciones que se realizarán en cada fase dependerá del tamaño del proyecto. La Figura 6 resume este proceso.

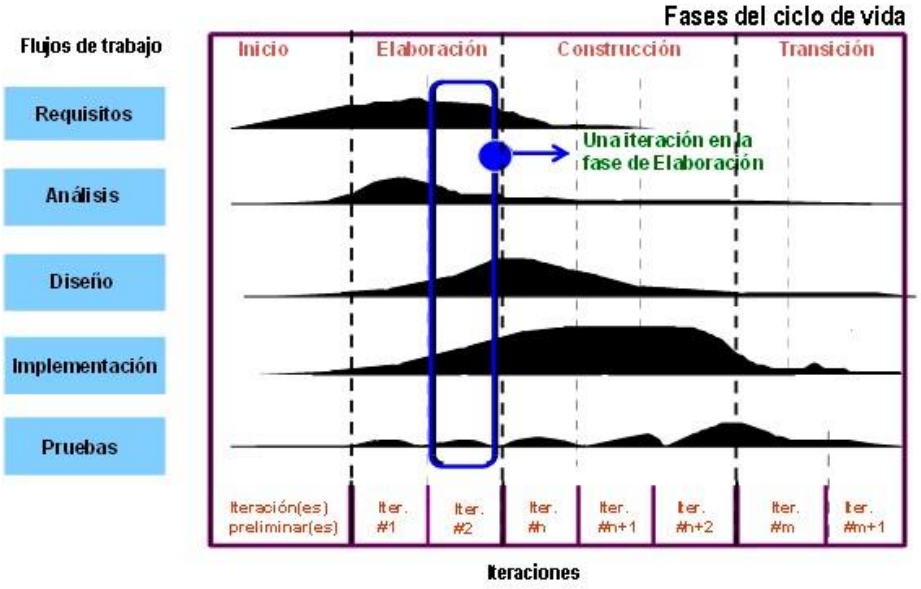


Figura 6. Fases del ciclo de vida del proceso unificado

El Proceso Unificado de Rational

Rational Unified Process (RUP) (Kruchten, 1998) es un producto comercial de IBM *Rational Software* que constituye un ejemplo de modelo de proceso resultado de la colaboración de UML (Booch, Rumbaugh y Jacobson, 2005) y el USDP (Jacobson, Booch y Rumbaugh, 2000). En una palabra, RUP es una variante o un producto comercial del UP descrito anteriormente.

RUP defiende que los modelos de proceso convencionales presentan una única perspectiva del proceso, es decir, una secuencia de actividades, conectadas, que incorporan estrategias para llevar a cabo la evolución del *software*. Sin embargo, RUP, de acuerdo a Sommerville (2005, 76), se puede describir bajo tres perspectivas distintas:

- **Perspectiva estática.** La perspectiva estática muestra cómo están dispuestas las actividades (lo que se conoce como **flujos de trabajo** en terminología RUP) que comprende el proceso: (i)
  - **Modelado del negocio**, donde se modelan los procesos de negocio mediante los casos de uso de negocio.
  - **Requisitos**, donde se identifican los actores que interactúan con el sistema y se desarrollan casos de uso que modelan los requisitos del sistema.
  - **Análisis y diseño**, donde se genera y se documenta un modelo de diseño utilizando modelos arquitectónicos, modelos de componentes, modelos de objetos y modelos de secuencia.
  - **Implementación, pruebas, despliegue, gestión de la configuración y del cambio, gestión del proyecto, y entorno**, donde se pone a disposición del equipo de desarrollo un conjunto de herramientas software que ayudarán a la implementación del sistema.
- **Perspectiva dinámica**. La perspectiva dinámica muestra las fases del modelo de proceso a lo largo del tiempo.
- **Perspectiva práctica**. La perspectiva práctica sugiere buenas prácticas que se deben utilizar durante el proceso.

La distinción entre fases y flujos de trabajo (actividades), y el reconocimiento del despliegue del *software* como parte del proceso de desarrollo, son algunas de las innovaciones más importantes que merece la pena destacar de RUP. Las fases son dinámicas y tienen objetivos. Los flujos de trabajo son estáticos y se corresponden con actividades que no están asociadas a una única fase, pero que se pueden utilizar a lo largo del proceso de desarrollo para alcanzar los objetivos de cada fase.

La Figura 7 muestra las distintas fases de RUP. Cada fase se puede llevar a cabo de modo iterativo, donde cada resultado se desarrolla de manera incremental. Por otro lado, el conjunto de todas las fases (es decir, un ciclo) también se puede llevar a cabo de manera incremental, tal y como muestra la flecha en la Figura 7, que parte de la fase de transición y vuelve hacia la fase de inicio.

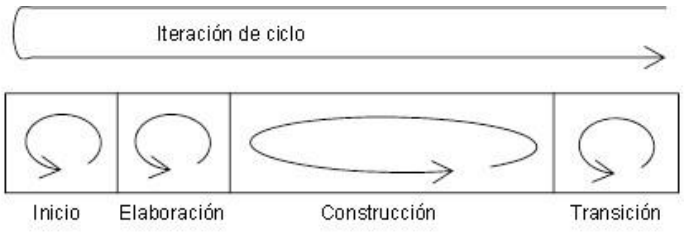


Figura 7. Fases en el Proceso Unificado de Rational

Con lo cual, el sistema *software* final estará constituido por un conjunto de artefactos. Los distintos artefactos normalmente compartirán los mismos conceptos del dominio del problema, pero a un nivel diferente de abstracción y bajo dimensiones distintas como se verá posteriormente. Por ejemplo, el término **transacción** que aparece en el documento de requisitos de usuario y la clase **Transacción** que aparece tanto en el diseño como en el código describen el mismo concepto «transacción» (Tarr et al., 1999).

Bajo el enfoque de Rational, el proyecto *software* será llevado a cabo por un conjunto de actores que de cara al sistema podrán adoptar distintos roles. Cada rol participará en una o más actividades del proyecto y entre todos producirán los distintos artefactos. De esta manera, actividades, roles y artefactos constituyen los elementos de proceso básicos de RUP.

Durante la **fase de inicio**, se desarrolla una descripción del sistema final y se presenta un análisis de negocio para el sistema.

Durante la **fase de elaboración**, se especifican en detalle la mayoría de los casos de uso del producto y se diseña la arquitectura del sistema. En RUP, la arquitectura se expresa en forma de vistas a través de modelos del sistema (o modelos *software*), los cuales, todos juntos, representan el sistema completo. Con lo cual, se tendrán los siguientes modelos que expresan las distintas vistas arquitectónicas: modelo de casos de uso (aunque en la terminología clásica el modelo de casos de uso formaría parte del modelo de análisis), modelo de análisis, modelo de diseño, modelo de implementación y modelo de despliegue. Además, en esta fase se realizan los casos de uso más críticos que se identificaron en la fase de inicio.

Durante la **fase de construcción** se crea el sistema. Al final de esta fase el sistema contiene todos los casos de uso que se acordaron para el desarrollo de esta versión. Sin embargo, no se garantiza todavía que el sistema final se encuentre libre de errores.

La **fase de transición** cubre el periodo durante el cual el producto constituye una **versión *beta*** del sistema (es decir, una versión que no se considera final, pero que ya se puede probar y entregar como una primera aproximación). Así, en la versión *beta* un número reducido de usuarios prueba el sistema e informa de los problemas encontrados. La fase de transición incluye por tanto actividades del tipo: formación del cliente, asistencia técnica y corrección de errores.

Las buenas prácticas recomendadas por RUP para su utilización en el desarrollo de sistemas son las que se describen a continuación (Sommerville, 2005, 78):

- **Desarrollar *software* de manera iterativa.** Planificar los distintos incrementos del sistema basándose en las prioridades del cliente, y desarrollar y entregar las características del sistema que tienen mayor prioridad al comienzo del proceso de desarrollo.
- **Gestión de requisitos.** Documentar todos los requisitos del cliente y llevar el control de todos los cambios que se producen en cada requisito. Analizar el impacto de los cambios sobre el sistema antes de aceptarlos.
- **Hacer uso de arquitecturas basadas en componentes.** Estructurar la arquitectura del sistema en forma de componentes.
- **Modelado visual del *software*.** Utilizar modelos gráficos UML para presentar las vistas estáticas y dinámicas del sistema.
- **Verificar la calidad del sistema.** Comprobar que el *software* cumple los estándares de calidad de la organización.
- **Controlar los cambios que se producen en el *software*.** Utilizar herramientas y procedimientos de gestión de la configuración y gestión del cambio para controlar las modificaciones que se producen en el sistema.

Si bien es cierto que RUP no resulta adecuado en todos los proyectos de desarrollo de *software*, sí que representa una aproximación adecuada para procesos genéricos. En este sentido, RUP constituye un marco de trabajo (*framework*) con un conjunto más o menos completo de elementos de proceso que tiene que ser adaptado a cada caso (es decir, adaptado en función del tamaño del sistema, el dominio en el cual va a funcionar, su complejidad, las capacidades del personal de la organización, etc.) (Jacobson, Booch y

Rumbaugh, 2000), ya que no todos los proyectos necesitan un conjunto completo de elementos. Con lo cual, antes de aplicar RUP a los procesos de una organización se deberá pensar en lo que realmente se necesita y qué cosas se podrían descartar (Hanssen, Westerheim y Bjørson, 2005).

Utilizar RUP como base para los procesos de desarrollo de una organización implica, aparte de la adaptación del modelo, conservar las propiedades básicas de RUP: documentación y diálogo con el cliente basado en casos de uso, un enfoque arquitectónico, con procesos iterativos y un desarrollo incremental del producto.

Sin embargo, no se encuentran en RUP muchas directrices que indiquen cómo hay que hacer esta adaptación. Hanssen, Westerheim y Bjørson (2005) realizan un estudio y describen tres formas de adaptar RUP: (i) el *framework* se adapta a cada proyecto, de manera individual, lo que supone una gran cantidad de trabajo; (ii) el *framework* se adapta generando otro *framework* que será un subconjunto de RUP, pero ahora en sintonía con las características generales de la organización; y (iii) la organización identifica y describe los distintos tipos de proyecto que abarca, y el *framework* se adapta a cada uno de los tipos que se han encontrado. Ahora bien, independientemente de la técnica de adaptación utilizada, en un último paso, siempre habrá que realizar una adaptación final al caso (proyecto), cuya complejidad dependerá de la técnica específica adoptada que se supone estará más acorde con la organización.

A lo largo del tiempo se han detectado diversos problemas y limitaciones en RUP, reconocido por los mismos autores que lo crearon, y es por ello que uno de ellos, Ivar Jacobson ideó otro enfoque con el objetivo de solventar y mejorar el modelo proceso propuesto por RUP. Esta nueva aproximación se conoce como *Essential Unified Process* (EssUP) y recoge diversas técnicas de RUP, del modelo de mejora de procesos *Capability Maturity Model Integration* (CMMI) y del desarrollo ágil para su aplicación en los proyectos *software*. Lo que propone EssUP es que la organización pueda tomar aquellas prácticas que considere útiles para un proyecto *software* en particular, y las combine como estime necesario dentro del proceso de desarrollo de *software*, de manera aislada si así se requiere, y no incluyendo todas las que a su vez pueda tener relacionadas de acuerdo a RUP, cuyo volumen y esfuerzo podría llegar a ser inviable para el proyecto.

2.5. Referencias bibliográficas

Boehm, B. (1988). A Spiral Model of Software Development and Enhancement. *Journal of IEEE Computer,* 21 (5), 61-72.

Booch, G., Rumbaugh, J. y Jacobson, I. (2005). *Unified Modeling Language User Guide. 2nd Edition*. Addison-Wesley.

Hanssen, G.K., Westerheim, H. y Bjørson, F.O. (2005). Tailoring RUP to a defined project type: A case study. *Actas de 6th International Conference on Product-Focused Software Process Improvement,* PROFES, Oulu, Finland. Springer-Verlag, 3547, 314327.

Heineman, G. y Council, W. (2001). *Component-Based Software Engineering - Putting the Pieces Together*. Addison-Wesley.

Humphrey, W.S. (1990). *Managing the Software Process*. Addison-Wesley.

Jacobson , I., Booch, G. y Rumbaugh, J. (2000). *El Proceso Unificado de Desarrollo de Software*. Addison Wesley.

Jones, S. (2005). Toward an acceptable definition of service. *IEEE Software*, 22(3), 87–93.

Kruchten, P. (1998). *Rational Unified Process*. Addison-Wesley.

Microsoft (2002). *Arquitectura de aplicaciones para .NET. Diseño de aplicaciones y servicios*. McGraw-Hill.

Microsoft (2009). *Microsoft® Application Achitecture Guide. 2nd Edition*. Microsoft Corporation.

Stevens, P. y Pooley, R. (2000). *Using UML. Software engineering with objects and components. Updated edition*. Addison Wesley.

Pressman, R. S. (2010). *Ingeniería del software. Un enfoque práctico. Séptima edición*. McGraw-Hill.

Sommerville, I. (2005). *Ingeniería del Software. Séptima edición*. España: Pearson Addison-Wesley.

Tarr, P., Ossher, H., Harrison, W. y Sutton, S.M. (1999). N degrees of separation: Multidimensional separation of concerns. *Actas de 21st International Conference on Software Engineering (ICSE'99)*, 107-119.