



INGENIERÍA DE SOFTWARE AVANZADA

TEMA 2

CLAVE: MIS 410

PROFESOR: M.C. ALEJANDRO GUTIÉRREZ DÍAZ

2. EL MARCO CONCEPTUAL

2.1 Metodología

2.2 UML

2.3 RUP

2.4 MDA

2.1 Metodología

Software

La palabra «software» se refiere al **equipamiento lógico** o **soporte lógico** de un computador digital, y comprende el conjunto de los componentes lógicos necesarios para hacer posible la realización de una tarea específica, en contraposición a los componentes físicos del sistema (hardware).

Tales componentes lógicos incluyen, entre otros, aplicaciones informáticas tales como procesador de textos, que permite al usuario realizar todas las tareas concernientes a edición de textos; software de sistema, tal como un sistema operativo, el que, básicamente, permite al resto de los programas funcionar adecuadamente, facilitando la interacción con los componentes físicos y el resto de las aplicaciones, también provee

Software es lo que se denomina **producto** en la Ingeniería de Software.

Definición de Software

Probablemente la definición más formal de software sea la siguiente:

Es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación. Extraído del estándar 729 del IEEE

Bajo esta definición, el concepto de software va más allá de los programas de cómputo en sus distintos estados: código fuente, binario o ejecutable; también su documentación, datos a procesar e información de usuario es parte del software: es decir, abarca todo lo intangible, todo lo "no físico" relacionado.

El término «software» fue usado por primera vez en este sentido por John W. Tukey en 1957. En las ciencias de la computación y la ingeniería de software, el software es toda la información procesada por los

sistemas informáticos: programas y datos. El concepto de leer diferentes secuencias de instrucciones desde la memoria de un dispositivo para controlar los cálculos fue introducido por Charles Babbage y a su socia, la matemática británica Augusta Ada Byron (1815-1852), hija del poeta inglés Lord Byron como parte de su máquina diferencial. La teoría que forma la base de la mayor parte del software moderno fue propuesta por vez primera por Alan Turing en su ensayo de 1936, "Los números computables", con una aplicación al problema de decisión.

Clasificación del software

Si bien esta distinción es, en cierto modo, arbitraria, y a veces confusa, a los fines prácticos se puede clasificar al software en tres grandes tipos:

- **Software de sistema:** Su objetivo es desvincular adecuadamente al usuario y al programador de los detalles del computador en particular que se use, aislándolo especialmente del procesamiento referido a las características internas de: memoria, discos, puertos y dispositivos de comunicaciones, impresoras, pantallas, teclados, etc. El software de sistema le procura al usuario y programador adecuadas interfaces de alto nivel, herramientas y utilidades de apoyo que permiten su mantenimiento. Incluye entre otros:
 - Sistemas operativos
 - Controladores de dispositivo
 - Herramientas de diagnóstico
 - Herramientas de Corrección y Optimización
 - Servidores
 - Utilidades
- **Software de programación:** Es el conjunto de herramientas que permiten al programador desarrollar programas informáticos, usando diferentes alternativas y lenguajes de programación, de una manera práctica. Incluye entre otros:
 - Editores de texto
 - Compiladores
 - Intérpretes
 - Enlazadores
 - Depuradores
 - Entornos de Desarrollo Integrados (IDE): Agrupan las anteriores herramientas, usualmente en un entorno visual, de forma que el programador no necesite introducir múltiples comandos para compilar, interpretar, depurar, etc. Habitualmente cuentan con una avanzada interfaz gráfica de usuario (GUI).
- **Software de aplicación:** Aquel que permite a los usuarios llevar a cabo una o varias tareas específicas, en cualquier campo de actividad susceptible de ser automatizado o asistido, con especial énfasis en los negocios. Incluye entre otros:

- Aplicaciones de Sistema de control y automatización industrial
- Aplicaciones ofimáticas
- Software educativo
- Software empresarial
- Bases de datos
- Telecomunicaciones (p.ej. internet y toda su estructura lógica)
- Videojuegos
- Software médico
- Software de Cálculo Numérico
- Software de Diseño Asistido (CAD)
- Software de Control Numérico (CAM)

Proceso de creación de software

Se define como Proceso al conjunto ordenado de pasos a seguir para llegar a la solución de un problema u obtención de un producto, en este caso particular, para lograr la obtención de un producto software que resuelva un problema.

Ese proceso de creación de software puede llegar a ser muy complejo, dependiendo de su porte, características y criticidad del mismo. Por ejemplo la creación de un sistema operativo es una tarea que requiere proyecto, gestión, numerosos recursos y todo un equipo disciplinado de trabajo. En el otro extremo, si se trata de un sencillo programa (ejemplo: resolución de una ecuación de segundo orden), éste puede ser realizado por un solo programador (incluso aficionado) fácilmente.

Es así que normalmente se dividen en tres categorías según su tamaño (líneas de código) y/o costo: de Pequeño, Mediano y Gran porte. Existen varias metodologías para **estimarlo**, una de las más populares es el sistema COCOMO que provee métodos y un software (programa) que calcula y provee una estimación de todos los costos de producción en un "proyecto software" (relación horas/hombre, costo monetario, cantidad de líneas fuente de acuerdo a lenguaje usado, etc.).

Considerando los de gran porte, es necesario realizar tantas y tan complejas tareas, tanto técnicas, de gerenciamiento, fuerte gestión y análisis diversos (entre otras) que toda una ingeniería hace falta para su estudio y realización: es la Ingeniería de Software.

En tanto que en los de mediano porte, pequeños equipos de trabajo (incluso un avezado analista-programador solitario) puede realizar la tarea. Aunque, siempre en casos de mediano y gran porte (y a veces también en algunos de pequeño porte, según su complejidad), se deben seguir ciertas etapas que son necesarias para la construcción del

software. Tales etapas, si bien deben existir, son flexibles en su forma de aplicación, de acuerdo a la metodología o Proceso de Desarrollo escogido y utilizado por el equipo de desarrollo o analista-programador solitario (si fuere el caso).

Los **"procesos de desarrollo de software"** poseen reglas preestablecidas, y deben ser aplicados en la creación del software de mediano y gran porte, ya que en caso contrario lo más seguro es que el proyecto o no logre concluir o termine sin cumplir los objetivos previstos y con variedad de fallos inaceptables (fracasan, en pocas palabras). Entre tales "procesos" los hay ágiles o livianos (ejemplo XP), pesados y lentos (ejemplo RUP) y variantes intermedias; y normalmente se aplican de acuerdo al tipo y porte y tipología del software a desarrollar, a criterio del líder (si lo hay) del equipo de desarrollo. Algunos de esos procesos son Extreme Programming (XP), Rational Unified Process (RUP), Feature Driven Development (FDD), etc.

Cualquiera sea el "proceso" utilizado y aplicado en un desarrollo de software (RUP, FDD, etc.), y casi independientemente de él, siempre se debe aplicar un "Modelo de Ciclo de Vida".

Se estima que, del total de proyectos software grandes emprendidos, un 28% fracasan, un 46% caen en severas modificaciones que lo retrasan y un 26% son totalmente exitosos. Cuando un proyecto fracasa, rara vez es debido a fallas técnicas, la principal causa de fallos y fracasos es la falta de aplicación de una buena metodología o proceso de desarrollo.

Entre otras, una fuerte tendencia, desde hace pocas décadas, es mejorar las metodologías o procesos de desarrollo, o crear nuevas y concientizar a los profesionales en su utilización adecuada.

Normalmente los especialistas en el estudio y desarrollo de estas áreas (metodologías) y afines (tales como modelos y hasta la gestión misma de los proyectos) son los Ingenieros en Software, es su orientación. Los especialistas en cualquier otra área de desarrollo informático (analista, programador, Lic. en Informática, Ingeniero en Informática, Ingeniero de Sistemas, etc.) normalmente aplican sus conocimientos especializados pero utilizando modelos, paradigmas y procesos ya elaborados.

Es común para el desarrollo de software de mediano porte que los equipos humanos involucrados apliquen sus propias metodologías, normalmente un híbrido de los procesos anteriores y a veces con criterios propios.

El proceso de desarrollo puede involucrar numerosas y variadas tareas, desde lo administrativo, pasando por lo técnico y hasta la gestión y el

gerenciamiento. Pero casi rigurosamente siempre se cumplen ciertas **etapas mínimas**; las que se pueden resumir como sigue:

- Captura, Elicitación, Especificación y Análisis de requisitos (ERS)
- Diseño
- Codificación
- Pruebas (unitarias y de integración)
- Instalación y paso a Producción
- Mantenimiento

En las anteriores etapas pueden variar ligeramente sus nombres, o ser más globales, o contrariamente más refinadas; por ejemplo indicar como una única fase (a los fines documentales e interpretativos) de "Análisis y Diseño"; o indicar como "Implementación" lo que está dicho como "Codificación"; pero en rigor, todas existen e incluyen, básicamente, las mismas tareas específicas.

En el apartado 4 del presente artículo se brindan mayores detalles de cada una de las listadas etapas.

Modelos de Proceso o Ciclo de Vida

Para cada una de las fases o etapas listadas en el ítem anterior, existen sub-etapas (o tareas). El Modelo de Proceso o Modelo de Ciclo de Vida utilizado para el desarrollo define el orden para las tareas o actividades involucradas, también definen la coordinación entre ellas, enlace y realimentación entre las mencionadas etapas.

Entre los más conocidos se puede mencionar: **Modelo de prototipos**, Modelo en Cascada o secuencial, Modelo Espiral, Modelo Iterativo Incremental. De los antedichos hay a su vez algunas variantes o alternativas, más o menos atractivas según sea la aplicación requerida y sus requisitos.

Modelo de prototipos

En Ingeniería de software el **desarrollo con prototipos**, también llamado **modelo de prototipos** que pertenece a los **modelos de desarrollo evolutivo**, se inicia con la definición de los objetivos globales para el software, luego se identifican los requisitos conocidos y las áreas del esquema en donde es necesaria más definición. Entonces se plantea con rapidez una iteración de construcción de prototipos y se presenta el modelado (en forma de un diseño rápido).

El diseño rápido se centra en una representación de aquellos aspectos del software que serán visibles para el cliente o el usuario final (por ejemplo, la configuración de la interfaz con el usuario y el formato de los

despliegues de salida). El diseño rápido conduce a la construcción de un prototipo, el cual es evaluado por el cliente o el usuario para una retroalimentación; gracias a ésta se refinan los requisitos del software que se desarrollará.

La iteración ocurre cuando el prototipo se ajusta para satisfacer las necesidades del cliente. Esto permite que al mismo tiempo el desarrollador entienda mejor lo que se debe hacer y el cliente vea resultados a corto plazo.

Ventajas

- Este modelo es útil cuando el cliente conoce los objetivos generales para el software, pero no identifica los requisitos detallados de entrada, procesamiento o salida.
- También ofrece un mejor enfoque cuando el responsable del desarrollo del software está inseguro de la eficacia de un algoritmo, de la adaptabilidad de un sistema operativo o de la forma que debería tomar la interacción humano-máquina.

La construcción de prototipos se puede utilizar como un modelo del proceso independiente, se emplea más comúnmente como una técnica susceptible de implementarse dentro del contexto de cualquiera de los modelos del proceso expuestos.

Sin importar la forma en que éste se aplique, el paradigma de construcción de prototipos ayuda al desarrollador de software y al cliente a entender de mejor manera cuál será el resultado de la construcción cuando los requisitos estén satisfechos. De esta manera, este ciclo de vida en particular, involucra al cliente más profundamente para adquirir el producto.

Inconvenientes

- El usuario tiende a crearse unas expectativas cuando ve el prototipo de cara al sistema final. A causa de la intención de crear un prototipo de forma rápida, se suelen desatender aspectos importantes, tales como la calidad y el mantenimiento a largo plazo, lo que obliga en la mayor parte de los casos a reconstruirlo una vez que el prototipo ha cumplido su función. Es frecuente que el usuario se muestre reacio a ello y pida que sobre ese prototipo se construya el sistema final, lo que lo convertiría en un **prototipo evolutivo**, pero partiendo de un estado poco recomendado.
- En aras de desarrollar rápidamente el prototipo, el desarrollador suele tomar algunas decisiones de implementación poco convenientes (por ejemplo, elegir un lenguaje de programación incorrecto porque proporcione un desarrollo más rápido). Con el

paso del tiempo, el desarrollador puede olvidarse de la razón que le llevó a tomar tales decisiones, con lo que se corre el riesgo de que dichas elecciones pasen a formar parte del sistema final.

- por que es muy difícil y complejo realizarlo.

Conclusiones

A pesar de que tal vez surjan problemas, la construcción de prototipos puede ser un paradigma efectivo para la ingeniería del software. La clave es definir las reglas del juego desde el principio; es decir, el cliente y el desarrollador se deben poner de acuerdo en:

- Que el prototipo se construya y sirva como un **mecanismo para la definición de requisitos**.
- Que el prototipo se **descarte**, al menos en parte.
- Que después se desarrolle el software real con un enfoque hacia la **calidad**.

Modelo Cascada

Este, aunque es más comúnmente conocido como Modelo en cascada es también llamado "Modelo Clásico", "Modelo Tradicional" o "Modelo Lineal Secuencial".

El Modelo en cascada puro **difícilmente se utilice tal cual**, pues esto implicaría un previo y absoluto conocimiento de los requisitos, la no volatilidad de los mismos (o rigidez) y etapas subsiguientes libres de errores; ello sólo podría ser aplicable a escasos y pequeños desarrollos de sistemas.

En estas circunstancias, el paso de una etapa a otra de las mencionadas sería sin retorno, por ejemplo pasar del Diseño a la Codificación implicaría un diseño exacto y sin errores ni probable modificación o evolución: "codifique lo diseñado que no habrán en absoluto variantes ni errores". Esto es utópico; ya que intrínsecamente el software es de carácter evolutivo, cambiante y difícilmente libre de errores, tanto durante su desarrollo como durante su vida operativa.

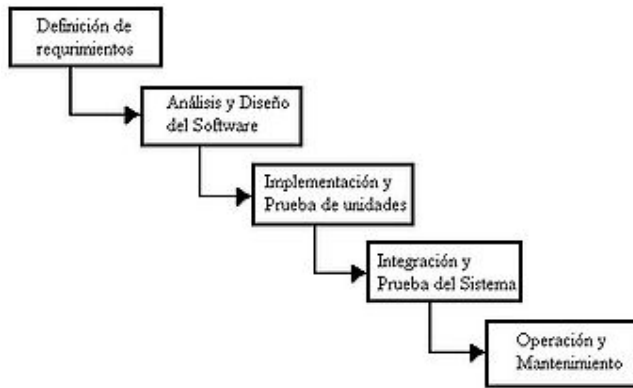


Fig. 2 - Modelo Cascada Puro o Secuencial para el Ciclo de Vida del Software

Algún cambio durante la ejecución de una cualquiera de las etapas en este modelo secuencial implicaría reiniciar desde el principio todo el ciclo completo, lo cual redundaría en altos costos de tiempo y desarrollo. La figura 2 muestra un posible esquema del modelo en cuestión.

Sin embargo, **el modelo cascada en algunas de sus variantes es uno de los actualmente más utilizados**, por su eficacia y simplicidad, más que nada en software de pequeño y algunos de mediano porte; pero nunca (o muy rara vez) se lo usa en su forma pura, como se dijo anteriormente.

En lugar de ello, siempre se produce alguna realimentación entre etapas, que no es completamente predecible ni rígida; esto da oportunidad al desarrollo de productos software en los cuales hay ciertas incertidumbres, cambios o evoluciones durante el ciclo de vida.

Así por ejemplo, una vez capturados (elicitados) y especificados los requisitos (primera etapa) se puede pasar al diseño del sistema, pero durante esta última fase lo más probable es que se deban realizar ajustes en los requisitos (aunque sean mínimos), ya sea por fallas detectadas, ambigüedades o bien por que los propios requisitos han cambiado o evolucionado; con lo cual **se debe retornar a la primera o previa etapa**, hacer los pertinentes reajustes y luego continuar nuevamente con el diseño; esto último se conoce como realimentación.

Lo normal en el modelo cascada será entonces la aplicación del mismo con sus etapas realimentadas de alguna forma, permitiendo retroceder de una a la anterior (e incluso poder saltar a varias anteriores) si es requerido.

De esta manera se obtiene un "**Modelo Cascada Realimentado**", que puede ser esquematizado como lo ilustra la figura 3.

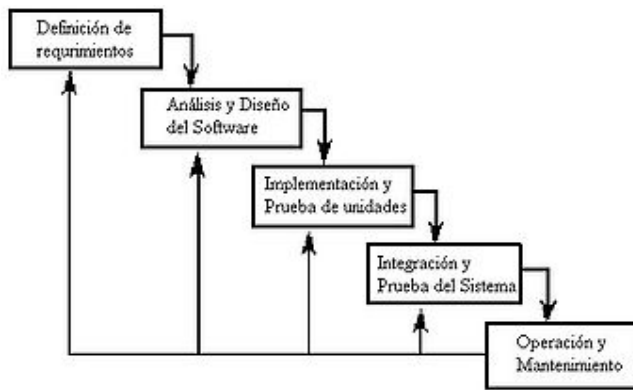


Fig. 3 - Modelo Cascada Realimentado para el Ciclo de Vida

Lo dicho es, a grandes rasgos, la forma y utilización de este modelo, uno de los más usados y populares. El modelo Cascada Realimentado resulta muy atractivo, hasta ideal, si el proyecto presenta alta rigidez (pocos o ningún cambio, no evolutivo), los requisitos son muy claros y están correctamente especificados.

Hay más variantes similares al modelo: refino de etapas (más etapas, menores y más específicas) e incluso mostrar menos etapas de las indicadas, aunque en tal caso la faltante estará dentro de alguna otra. El orden de esas fases indicadas en el ítem previo es el lógico y adecuado, pero adviértase, como se dijo, que normalmente habrá realimentación hacia atrás.

El modelo lineal o en Cascada es el paradigma más antiguo y extensamente utilizado, sin embargo las críticas a él (ver desventajas) han puesto en duda su eficacia. **Pese a todo** tiene un lugar muy importante en la Ingeniería de software y **continúa siendo el más utilizado**; y siempre es mejor que un enfoque al azar.

Desventajas del Modelo Cascada:

- Los cambios introducidos durante el desarrollo pueden confundir al equipo profesional en las etapas tempranas del proyecto. Si los cambios se producen en etapa madura (codificación o prueba) pueden ser catastróficos para un proyecto grande.
- No es frecuente que el cliente o usuario final explicita clara y completamente los requisitos (etapa de inicio); y el modelo lineal lo requiere. La incertidumbre natural en los comienzos es luego difícil de acomodar.
- El cliente debe tener paciencia ya que el software no estará disponible hasta muy avanzado el proyecto. Un error detectado por

el cliente (en fase de operación) puede ser desastroso, implicando reinicio del proyecto con altos costos.

Modelos Evolutivos

El software evoluciona con el tiempo. Los requisitos del usuario y del producto suelen cambiar conforme se desarrolla el mismo. Las fechas de mercado y la competencia hacen que no sea posible esperar a poner en el mercado un producto absolutamente completo, por lo que se debe introducir una versión funcional limitada de alguna forma para aliviar las presiones competitivas.

En esas u otras situaciones similares los desarrolladores necesitan modelos de progreso que estén diseñados para acomodarse a una evolución temporal o progresiva, donde los requisitos centrales son conocidos de antemano, aunque no estén bien definidos a nivel detalle.

En el modelo Cascada y Cascada Realimentado no se tiene en cuenta la naturaleza evolutiva del software, se plantea como estático con requisitos bien conocidos y definidos desde el inicio.

Los evolutivos son modelos iterativos, permiten desarrollar versiones cada vez más completas y complejas, hasta llegar al objetivo final deseado; incluso evolucionar más allá, durante la fase de operación.

Los modelos “Iterativo Incremental” y “Espiral” (entre otros) son dos de los más conocidos y utilizados del tipo evolutivo.

Modelo Iterativo Incremental

En términos generales, podemos distinguir, en la figura 4, los pasos generales que sigue el proceso de desarrollo de un producto software.

En el modelo de ciclo de vida seleccionado, se identifican claramente dichos pasos. La Descripción del Sistema es esencial para especificar y confeccionar los distintos incrementos hasta llegar al Producto global y final. Las actividades concurrentes (Especificación, Desarrollo y Validación) sintetizan el desarrollo pormenorizado de los incrementos, que se hará posteriormente.



Fig. 4 - Diagrama genérico del Desarrollo Evolutivo Incremental

El diagrama 4 nos muestra en forma muy esquemática, el funcionamiento de un Ciclo Iterativo Incremental, el cual permite la entrega de versiones parciales a medida que se va construyendo el producto final. Es decir, a medida que cada incremento definido llega a su etapa de operación y mantenimiento. Cada versión emitida incorpora a los anteriores incrementos las funcionalidades y requisitos que fueron analizados como necesarios. *El Incremental es un **modelo de tipo evolutivo** que está basado en varios ciclos Cascada realimentados aplicados repetidamente, con una filosofía iterativa.*

En la figura 5 se muestra un refinamiento del diagrama previo, bajo un esquema temporal, para obtener finalmente el esquema del Modelo de ciclo de vida Iterativo Incremental, con sus actividades genéricas asociadas. Aquí se observa claramente cada ciclo cascada que es aplicado para la obtención de un incremento; estos últimos se van integrando para obtener el producto final completo. Cada incremento es un ciclo Cascada Realimentado, aunque, por simplicidad, en la figura 5 se muestra como secuencial puro.

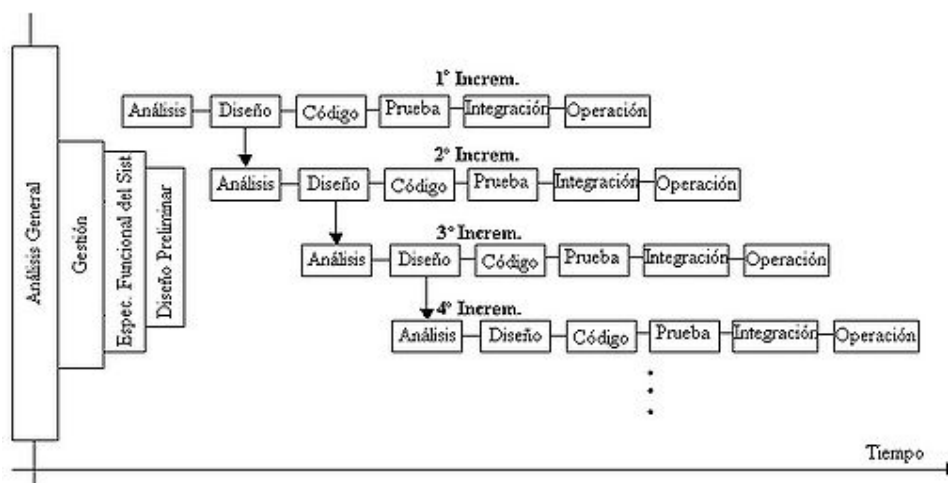


Fig. 5 - Modelo Iterativo Incremental para el ciclo de vida del software

Se observa que existen actividades de desarrollo (para cada incremento) que son realizadas en paralelo o concurrentemente, así por ejemplo, en la figura, mientras se realiza el diseño detalle del primer incremento ya se está realizando en análisis del segundo.

La figura 5 es sólo esquemática, un incremento no necesariamente se iniciará durante la fase de diseño del anterior, puede ser posterior (incluso antes), en cualquier tiempo de la etapa previa. Cada incremento concluye con la actividad de "Operación y Mantenimiento" (indicada "Operación" en la figura), que es donde se produce la entrega del producto parcial al cliente.

El momento de inicio de cada incremento es dependiente de varios factores: tipo de sistema; independencia o dependencia entre incrementos (dos de ellos totalmente independientes pueden ser fácilmente iniciados al mismo tiempo si se dispone de personal suficiente); capacidad y cantidad de profesionales involucrados en el desarrollo; etc.

Bajo este modelo se entrega software "por partes funcionales más pequeñas", pero reutilizables, llamadas incrementos. En general cada incremento se construye sobre aquel que ya fue entregado.

Como se muestra en la figura 5, se aplican secuencias Cascada en forma escalonada, mientras progresa el tiempo calendario. Cada secuencia lineal o Cascada produce un incremento y a menudo el primer incremento es un sistema básico, con muchas funciones suplementarias (conocidas o no) sin entregar.

El cliente utiliza inicialmente ese sistema básico intertanto, el resultado de su uso y evaluación puede aportar al plan para el desarrollo del/los siguientes incrementos (o versiones). Además también aportan a ese plan otros factores, como lo es la priorización (mayor o menor urgencia en la necesidad de cada incremento) y la dependencia entre incrementos (o independencia).

Luego de cada integración se entrega un producto con mayor funcionalidad que el previo. El proceso se repite hasta alcanzar el software final completo.

Siendo iterativo, *con el modelo Incremental se entrega un producto parcial pero completamente operacional en cada incremento*, y no una parte que sea usada para reajustar los requerimientos (como si ocurre en el Modelo de Construcción de Prototipos).

El enfoque Incremental resulta muy útil con baja dotación de personal para el desarrollo; también si no hay disponible fecha límite del proyecto

por lo que se entregan versiones incompletas pero que proporcionan al usuario funcionalidad básica (y cada vez mayor). También es un modelo útil a los fines de evaluación.

Nota: Puede ser considerado y útil, en cualquier momento o incremento incorporar temporalmente el paradigma MCP como complemento, teniendo así una mixtura de modelos que mejoran el esquema y desarrollo general.

Ejemplo:

Un procesador de texto que sea desarrollado bajo el paradigma Incremental podría aportar, en principio, funciones básicas de edición de archivos y producción de documentos (algo como un editor simple). En un segundo incremento se le podría agregar edición más sofisticada, y de generación y mezcla de documentos. En un tercer incremento podría considerarse el agregado de funciones de corrección ortográfica, esquemas de paginado y plantillas; en un cuarto capacidades de dibujo propias y ecuaciones matemáticas. Así sucesivamente hasta llegar al procesador final requerido. Así, el producto va creciendo, acercándose a su meta final, pero desde la entrega del primer incremento ya es útil y funcional para el cliente, el cual observa una respuesta rápida en cuanto a entrega temprana; sin notar que la fecha límite del proyecto puede no estar acotada ni tan definida, lo que da margen de operación y alivia presiones al equipo de desarrollo.

Como se dijo, el Iterativo Incremental es un modelo del tipo evolutivo, es decir donde se permiten y esperan probables cambios en los requisitos en tiempo de desarrollo; se admite cierto margen para que el software pueda evolucionar. Aplicable cuando los requisitos son medianamente bien conocidos pero no son completamente estáticos y definidos, cuestión esa que si es indispensable para poder utilizar un modelo Cascada.

El modelo es aconsejable para el desarrollo de software en el cual se observe, en su etapa inicial de análisis, que posee áreas bastante bien definidas a cubrir, con suficiente independencia como para ser desarrolladas en etapas sucesivas.

Tales áreas a cubrir suelen tener distintos grados de apremio por lo cual las mismas se deben priorizar en un análisis previo, es decir, definir cual será la primera, la segunda, y así sucesivamente; esto se conoce como "definición de los incrementos" en base a priorización.

Pueden no existir prioridades funcionales por parte del cliente, pero el desarrollador debe fijarlas de todos modos y con algún criterio, ya que en base a ellas se desarrollarán y entregarán los distintos incrementos.

El hecho de que existan incrementos funcionales del software lleva inmediatamente a pensar en un **esquema de desarrollo modular**, por tanto este modelo facilita tal paradigma de diseño.

En resumen, un modelo Incremental lleva a pensar en un desarrollo modular, con entregas parciales del producto software denominados “incrementos” del sistema, que son escogidos en base a prioridades predefinidas de algún modo.

El modelo permite una implementación con refinamientos sucesivos (ampliación y/o mejora). Con cada incremento se agrega nueva funcionalidad o se cubren nuevos requisitos o bien se mejora la versión previamente implementada del producto software.

Este modelo brinda cierta flexibilidad para que durante el desarrollo se incluyan cambios en los requisitos por parte del usuario, un cambio de requisitos propuesto y aprobado puede analizarse e implementarse como un nuevo incremento o, eventualmente, podrá constituir una mejora/ajuste de uno ya planeado.

Aunque si se produce un cambio de requisitos por parte del cliente que afecte incrementos previos ya terminados (detección/incorporación tardía) *se debe evaluar la factibilidad y realizar un acuerdo con el cliente, ya que puede impactar fuertemente en los costos.*

La selección de este modelo permite realizar **entregas funcionales tempranas al cliente** (lo cual es beneficioso tanto para él como para el grupo de desarrollo). Se priorizan las entregas de aquellos módulos o incrementos en que surja la necesidad operativa de hacerlo, por ejemplo para cargas previas de información, indispensable para los incrementos siguientes.

El modelo Iterativo Incremental no obliga a especificar con precisión y detalle absolutamente todo lo que el sistema debe hacer, (y cómo), antes de ser construido (como el caso del cascada, con requisitos congelados). Sólo se hace en el incremento en desarrollo. Esto torna más manejable el proceso y reduce el impacto en los costos.

Esto es así, porque en caso de alterar o rehacer los requisitos, solo afecta una parte del sistema. Aunque, lógicamente, esta situación se agrava si se presenta en estado avanzado, es decir en los últimos incrementos. *En definitiva, el modelo facilita la incorporación de nuevos requisitos durante el desarrollo.*

Con un paradigma Incremental se reduce el tiempo de desarrollo inicial, ya que se implementa funcionalidad parcial. También provee un impacto ventajoso frente al cliente, que es la entrega temprana de partes operativas del software.

El modelo proporciona todas las ventajas del modelo en cascada realimentado, reduciendo sus desventajas sólo al ámbito de cada incremento.

El modelo Incremental **no es recomendable** para casos de sistemas de tiempo real, de alto nivel de seguridad, de procesamiento distribuido, y/o de alto índice de riesgos.

Modelo Espiral

El Modelo Espiral fue propuesto inicialmente por Barry Boehm. Es un modelo evolutivo que conjuga la naturaleza iterativa del modelo MCP con los aspectos controlados y sistemáticos del Modelo Cascada. Proporciona potencial para desarrollo rápido de versiones incrementales.

En el modelo Espiral el software se construye en una serie de versiones incrementales. En las primeras iteraciones la versión incremental podría ser un modelo en papel o bien un prototipo. En las últimas iteraciones se producen versiones cada vez más completas del sistema diseñado

El modelo se divide en un número de Actividades de marco de trabajo, llamadas "**regiones de tareas**". En general existen entre tres y seis regiones de tareas (hay variantes del modelo). En la figura 6 se muestra el esquema de un Modelo Espiral con 6 regiones. En este caso se explica una variante del modelo original de Boehm, expuesto en su tratado de 1988; en 1998 expuso un tratado más reciente.

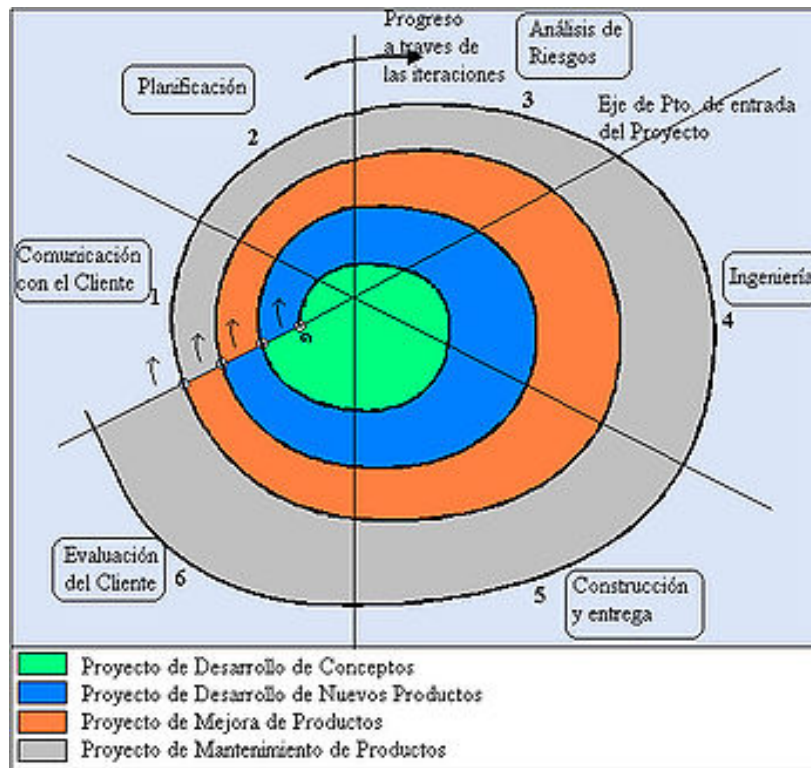


Fig. 6 - Modelo Espiral para el ciclo de vida del software

Las regiones definidas en el modelo de la figura son:

- Región 1 - Tareas requeridas para establecer la comunicación entre el cliente y el desarrollador.
- Región 2 - Tareas inherentes a la definición de los recursos, tiempo y otra información relacionada con el proyecto.
- Región 3 - Tareas necesarias para evaluar los riesgos técnicos y de gestión del proyecto.
- Región 4 - Tareas para construir una o más *representaciones* de la aplicación software.
- Región 5 - Tareas para construir la aplicación, instalarla, probarla y proporcionar soporte al usuario o cliente (Ej. documentación y práctica).
- Región 6 - Tareas para obtener la reacción del cliente, según la evaluación de lo creado e instalado en los ciclos anteriores.

Las *Actividades enunciadas para el marco de trabajo son generales y se aplican a cualquier proyecto*, grande, mediano o pequeño, complejo o no. Las regiones que definen esas actividades comprenden un "conjunto de tareas" del trabajo: ese conjunto **si** se debe adaptar a las características del proyecto en particular a emprender. Nótese que lo listado en los

ítems de 1 a 6 son conjuntos de tareas, algunas de las ellas normalmente dependen del proyecto o desarrollo en si.

Proyectos pequeños requieren baja cantidad de tareas y también de formalidad. En proyectos mayores o críticos cada región de tareas contiene labores de más alto nivel de formalidad. En cualquier caso se aplican actividades de protección (por ejemplo, gestión de configuración del software, garantía de calidad, etc.).

Al inicio del ciclo, o proceso evolutivo, el equipo de ingeniería gira alrededor del espiral (metafóricamente hablando) comenzando por el centro (marcado con \odot en la figura 6) y en el sentido indicado; el primer circuito de la espiral puede producir el desarrollo de una especificación del producto; los pasos siguientes podrían generar un prototipo y progresivamente versiones más sofisticadas del software.

Cada paso por la región de planificación provoca ajustes en el plan del proyecto; el coste y planificación se realimentan en función de la evaluación del cliente. El gestor de proyectos debe ajustar el número de iteraciones requeridas para completar el desarrollo.

El modelo espiral puede ir adaptándose y aplicarse a lo largo de todo el ciclo de vida del software (en el modelo clásico, o cascada, el proceso termina a la entrega del software).

Una visión alternativa del modelo puede observarse examinando el "eje de punto de entrada de proyectos". Cada uno de los circulitos (\odot) fijados a lo largo del eje representan puntos de arranque de los distintos proyectos (relacionados); a saber:

- Un proyecto de "Desarrollo de Conceptos" comienza al inicio de la espiral, hace múltiples iteraciones hasta que se completa, es la zona marcada con verde.
- Si lo anterior se va a desarrollar como producto real, se inicia otro proyecto: "Desarrollo de nuevo Producto". Que evolucionará con iteraciones hasta culminar; es la zona marcada en color azul.
- Eventual y análogamente se generarán proyectos de "Mejoras de Productos" y de "Mantenimiento de productos", con las iteraciones necesarias en cada área (zonas roja y gris, respectivamente).

Cuando la espiral se caracteriza de esta forma, está operativa **hasta que el software se retira**, eventualmente puede estar inactivo (el proceso), pero cuando se produce un cambio el proceso arranca nuevamente en el punto de entrada apropiado (por ejemplo, en "Mejora del Producto").

El modelo espiral da un enfoque realista, que evoluciona igual que el software; se adapta muy bien para desarrollos a gran escala.

El Espiral utiliza el MCP para reducir riesgos y permite aplicarlo en cualquier etapa de la evolución. Mantiene el enfoque clásico (cascada) pero incorpora un marco de trabajo iterativo que refleja mejor la realidad.

Este modelo *requiere considerar riesgos técnicos* en todas las etapas del proyecto; aplicado adecuadamente debe reducirlos antes de que sean un verdadero problema.

El Modelo evolutivo como el Espiral es particularmente apto para el desarrollo de Sistemas Operativos (complejos); también en sistemas de altos riesgos o críticos (Ej. navegadores y controladores aeronáuticos) y en todos aquellos en que sea necesaria una fuerte gestión del proyecto y sus riesgos, técnicos o de gestión.

Desventajas importantes:

- Requiere mucha experiencia y habilidad para la evaluación de los riesgos, lo cual es requisito para el éxito del proyecto.
- Es difícil convencer a los grandes clientes que se podrá controlar este enfoque evolutivo.

Este modelo no se ha usado tanto, como el Cascada (Incremental) o MCP, por lo que no se tiene bien medida su eficacia, es un paradigma relativamente nuevo y difícil de implementar y controlar.

Modelo Espiral Win & Win

Una variante interesante del Modelo Espiral previamente visto (Fig. 6) es el "Modelo Espiral Win-Win" (Barry Boehm). El Modelo Espiral previo (clásico) sugiere la comunicación con el cliente para fijar los requisitos, en que simplemente se pregunta al cliente qué necesita y él proporciona la información para continuar; pero esto es en un contexto ideal que rara vez ocurre. Normalmente cliente y desarrollador entran en una negociación, se negocia coste frente a funcionalidad, rendimiento, calidad, etc.

"Es así que la obtención de requisitos requiere una negociación, que tiene éxito cuando ambas partes ganan".

Las mejores negociaciones se fuerzan en obtener "Victoria & Victoria" (Win & Win), es decir que el cliente gane obteniendo el producto que lo satisfaga, y el desarrollador también gane consiguiendo presupuesto y fecha de entrega realista. Evidentemente, este modelo requiere fuertes habilidades de negociación.

El modelo Win-Win define un conjunto de actividades de negociación al principio de cada paso alrededor de la espiral; se definen las siguientes actividades:

- 1 - Identificación del sistema o subsistemas clave de los directivos (*) (saber qué quieren).
- 2 - Determinación de "condiciones de victoria" de los directivos (saber qué necesitan y los satisface)
- 3 - Negociación de las condiciones "victoria" de los directivos para obtener condiciones "Victoria & Victoria" (negociar para que ambos ganen).

(*) Directivo: Cliente escogido con interés directo en el producto, que puede ser premiado por la organización si tiene éxito o criticado si no.

El modelo Win & Win hace énfasis en la negociación inicial, también introduce 3 hitos en el proceso llamados "puntos de fijación", que ayudan a establecer la completitud de un ciclo de la espiral, y proporcionan hitos de decisión antes de continuar el proyecto de desarrollo del software.

Etapas en el Desarrollo de Software

Captura, Análisis y Especificación de requisitos

Al inicio de un desarrollo (no de un proyecto), esta es la primera fase que se realiza, y, según el modelo de proceso adoptado, puede casi terminar para pasar a la próxima etapa (caso de Modelo Cascada Realimentado) o puede hacerse parcialmente para luego retomarla (caso Modelo Iterativo Incremental u otros de carácter evolutivo).

En simple palabras y básicamente, durante esta fase, se adquieren, reúnen y especifican las características funcionales y no funcionales que deberá cumplir el futuro programa o sistema a desarrollar.

Las bondades de las características, tanto del sistema o programa a desarrollar, como de su entorno, parámetros no funcionales y arquitectura dependen enormemente de lo bien lograda que esté esta etapa. Esta es, probablemente, la de mayor importancia y una de las fases más difíciles de lograr **certeramente**, pues no es automatizable, no es muy técnica y depende en gran medida de la habilidad y experiencia del analista que la realice.

Involucra fuertemente al usuario o cliente del sistema, por tanto tiene matices muy subjetivos y es difícil de modelar con certeza y/o aplicar una técnica que sea "la más cercana a la adecuada" (de hecho no existe "la

estrictamente adecuada"). Si bien se han ideado varias metodologías, incluso software de apoyo, para captura, elicitación y registro de requisitos, no existe una forma infalible o absolutamente confiable, y deben aplicarse conjuntamente buenos criterios y mucho sentido común por parte del o los analistas encargados de la tarea; es fundamental también lograr una fluida y adecuada comunicación y comprensión con el usuario final o cliente del sistema.

El artefacto más importante resultado de la culminación de esta etapa es lo que se conoce como Especificación de Requisitos Software o simplemente documento **ERS**.

Como se dijo, la habilidad del analista para interactuar con el cliente es fundamental; lo común es que el cliente tenga un objetivo general o problema a resolver, no conoce en absoluto el área (informática), ni su jerga, ni siquiera sabe con precisión qué debería hacer el producto software (qué y cuantas funciones) ni, mucho menos, cómo debe operar.

En otros casos menos frecuentes, el cliente "piensa" que sabe precisamente lo que el software tiene que hacer, y generalmente acierta muy parcialmente, pero su empeñamiento entorpece la tarea de elicitación. El analista debe tener la capacidad para lidiar con este tipo de problemas, que incluyen relaciones humanas; tiene que saber ponerse al nivel del usuario para permitir una adecuada comunicación y comprensión.

Escasas son las situaciones en que el cliente sabe con certeza e incluso con completitud lo que requiere de su futuro sistema, este es el caso más sencillo para el analista.

Las tareas relativas a captura, elicitación, modelado y registro de requerimientos, además de ser sumamente importante, puede llegar a ser dificultosa de lograr acertadamente y llevar bastante tiempo relativo al proceso total del desarrollo; al proceso y metodologías para llevar a cabo este conjunto de actividades normalmente se las asume parte propia de la Ingeniería de Software, pero dada la antedicha complejidad, actualmente se habla de una Ingeniería en Requisitos⁵, aunque ella aún no existe formalmente.

Hay grupos de estudio e investigación, en todo el mundo, que están exclusivamente abocados a la idear modelos, técnicas y procesos para intentar lograr la correcta captura, análisis y registro de requerimientos. Estos grupos son los que normalmente hablan de la Ingeniería en Requisitos; es decir se plantea ésta como un área o disciplina pero no como una carrera universitaria en si misma.

Algunos requisitos no necesitan la presencia del cliente, para ser capturados y/o analizados; en ciertos casos los puede proponer el mismo analista o, incluso, adoptar unilateralmente decisiones que considera adecuadas (tanto en requerimientos funcionales como no funcionales). Por citar ejemplos probables:

Algunos requisitos sobre la arquitectura del sistema, requisitos no funcionales tales como los relativos al rendimiento, nivel de soporte a errores operativos, plataformas de desarrollo, relaciones internas o ligas entre la información (entre registros o tablas de datos) a almacenar en caso de bases o bancos de datos, etc. Algunos funcionales tales como opciones secundarias o de soporte necesarias para una mejor o más sencilla operatividad; etc.

La obtención de especificaciones a partir del cliente (u otros actores intervinientes) es un proceso humano muy interactivo e iterativo; normalmente a medida que se captura la información, se la analiza y realimenta con el cliente, refinándola, puliéndola y corrigiendo si es necesario; cualquiera sea el método de ERS utilizado. EL analista siempre debe llegar a conocer la temática y el problema a resolver, dominarlo, hasta cierto punto, hasta el ámbito que el futuro sistema a desarrollar lo abarque.

Por ello el analista debe tener alta capacidad para comprender problemas de muy diversas áreas o disciplinas de trabajo (que no son específicamente suyas); así por ejemplo, si el sistema a desarrollar será para gestionar información de una aseguradora y sus sucursales remotas, el analista se debe compenetrar en cómo ella trabaja y maneja su información, desde niveles muy bajos e incluso llegando hasta los gerenciales.

Dada a gran diversidad de campos a cubrir, los analistas suelen ser asistidos por especialistas, es decir gente que conoce profundamente el área para la cual se desarrollará el software; evidentemente una única persona (el analista) no puede abarcar tan vasta cantidad de áreas del conocimiento. En empresas grandes de desarrollo de productos software, es común tener analistas especializados en ciertas áreas de trabajo.

Contrariamente, no es problema del cliente, es decir él no tiene por qué saber nada de software, ni de diseños, ni otras cosas relacionadas; sólo se debe limitar a aportar objetivos, datos e información (de mano propia o de sus registros, equipos, empleados, etc.) al analista, y guiado por él, para que, en primera instancia, defina el "**Universo de Discurso**", y con posterior trabajo logre confeccionar el adecuado documento ERS.

Es bien conocida la presión que sufren los desarrolladores de sistemas informáticos para comprender y/o rescatar las necesidades de los

clientes/usuarios. Cuanto más complejo es el contexto del problema más difícil es lograrlo, a veces se fuerza a los desarrolladores a tener que convertirse en casi expertos de los dominios que analizan.

Cuando esto no sucede es muy probable que se genere un conjunto de requisitos ⁶ erróneos o incompletos y por lo tanto un producto de software con alto grado de desaprobación por parte de los clientes/usuarios y un altísimo costo de reingeniería y mantenimiento. *Todo aquello que no se detecte, o resulte mal entendido en la etapa inicial provocará un fuerte impacto negativo en los requisitos, propagando esta corriente degradante a lo largo de todo el proceso de desarrollo e **incrementando su perjuicio cuanto más tardía sea su detección*** (Bell y Thayer 1976) (Davis 1993).

Procesos, modelado y formas de elicitación de requisitos

Siendo que la captura, elicitación y especificación de requisitos, es una parte crucial en el proceso de desarrollo de software, ya que de esta etapa depende el logro de los objetivos finales previstos, se han ideado modelos y diversas metodologías de trabajo para estos fines. También existen herramientas software que apoyan las tareas relativas realizadas por el ingeniero en requisitos.

El estándar IEEE 830-1998 brinda una normalización de las "Prácticas Recomendadas para la Especificación de Requisitos Software" ⁷.

A medida que se obtienen los requisitos, normalmente se los va analizando, el resultado de este análisis, con o sin el cliente, se plasma en un documento, conocido como ERS o Especificación de Requisitos Software, cuya estructura puede venir definida por varios estándares, tales como CMM-I.

Un primer paso para realizar el relevamiento de información es el conocimiento y definición acertada lo que se conoce como "Universo de Discurso" del problema, que se define y entiende por:

Universo de Discurso (UdeD): es el contexto general en el cual el software deberá ser desarrollado y deberá operar. El UdeD incluye todas las fuentes de información y todas las personas relacionadas con el software. Esas personas son conocidas también como **actores** de ese universo. El UdeD es la realidad circunstanciada por el conjunto de objetivos definidos por quienes demandaron el software.

A partir de la extracción y análisis de información en su ámbito se obtienen todas las especificaciones necesarias y tipos de requisitos para el futuro producto software.

El objetivo de la Ingeniería de Requisitos (IR) es sistematizar el proceso de definición de requisitos permitiendo elicitar, modelar y analizar el problema, generando un compromiso entre los Ingenieros de Requisitos y los clientes/usuarios, ya que ambos participan en la generación y definición de los requisitos del sistema. La IR aporta un conjunto de métodos, técnicas y herramientas que asisten a los ingenieros de requisitos (analistas) para obtener requerimientos lo más seguros, veraces, completos y oportunos posibles, permitiendo básicamente:

- Comprender el problema
- Facilitar la obtención de las necesidades del cliente/usuario
- Validar con el cliente/usuario
- Garantizar las especificaciones de requisitos

Si bien existen diversas formas, modelos y metodologías para elicitar, definir y documentar requerimientos, no se puede decir que alguna de ellas sea mejor o peor que la otra, suelen tener muchísimo en común, y todas cumplen el mismo objetivo. Sin embargo, lo que sí se puede decir sin dudas es que es indispensable utilizar alguna de ellas para documentar las especificaciones del futuro producto software.

Así por ejemplo, hay un grupo de investigación argentino que desde hace varios años ha propuesto y estudia el uso del LEL (Léxico Extendido del Lenguaje) y Escenarios como metodología, aquí ⁸ se presenta una de las tantas referencias y bibliografía sobre ello. Otra forma, más ortodoxa, de capturar y documentar requisitos se puede obtener en detalle, por ejemplo, en el trabajo de la Universidad de Sevilla sobre "Metodología para el Análisis de Requisitos de Sistemas Software" ⁹.

En la Fig. 7 se muestra un esquema, más o menos riguroso, aunque no detallado, de los pasos y tareas a seguir para realizar la captura, análisis y especificación de requerimientos software. También allí se observa qué artefacto o documento se obtiene en cada etapa del proceso. En el diagrama no se explicita metodología o modelo a utilizar, sencillamente se pautan las tareas que deben cumplirse, de alguna manera.

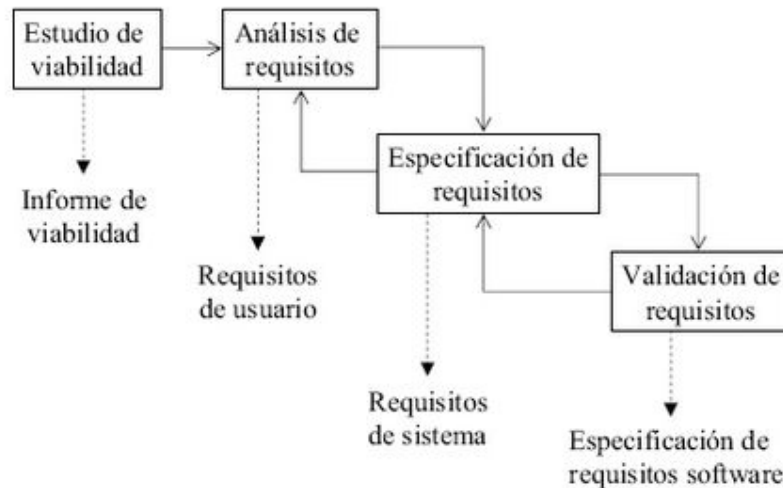


Fig. 7 - Diagrama de tareas para captura y análisis de Requisitos

Una posible lista, general y ordenada, de tareas recomendadas para obtener la definición de lo que se debe realizar, los productos a obtener y las técnicas a emplear durante la actividad de elicitación de requisitos, en fase de Especificación de Requisitos Software es:

- 1 - Obtener información sobre el dominio del problema y el sistema actual (UdeD).
- 2 - Preparar y realizar las reuniones para elicitación/negociación.
- 3 - Identificar/revisar los objetivos del usuario.
- 4 - Identificar/revisar los objetivos del sistema.
- 5 - Identificar/revisar los requisitos de información.
- 6 - Identificar/revisar los requisitos funcionales.
- 7 - Identificar/revisar los requisitos no funcionales.
- 8 - Priorizar objetivos y requisitos.

Algunos principios básicos a tener en cuenta:

- Presentar y entender cabalmente el dominio de la información del problema.
- Definir correctamente las funciones que debe realizar el Software.
- Representar el comportamiento del software a consecuencias de acontecimientos externos, particulares, incluso inesperados.
- Reconocer requisitos incompletos, ambiguos o contradictorios.
- Dividir claramente los modelos que representan la información, las funciones y comportamiento y características no funcionales.

Clasificación e identificación de requerimientos

Se pueden identificar **dos formas** de requisitos:

- Requisitos de usuario.

Los requisitos de usuario son frases en lenguaje natural junto a diagramas con los **servicios que el sistema** debe proporcionar, así como las restricciones bajo las que debe operar.

- Requisitos de sistema.

Los requisitos de sistema determinan los **servicios del sistema** y pero con las restricciones en detalle. Sirven como contrato.

Es decir, ambos son lo mismo, pero con distinto nivel de detalle.

- Ejemplo de requisito de usuario: El sistema debe hacer préstamos
- Ejemplo de requisito de sistema: Función préstamo: entrada código socio, código ejemplar; salida: fecha devolución; etc.

Se clasifican en tres los tipos de requisitos de sistema:

- 1 - Requisitos funcionales

Los requisitos funcionales describen:

- Los servicios que proporciona el sistema (funciones).
- La respuesta del sistema ante determinadas entradas.
- El comportamiento del sistema en situaciones particulares.

- 2 - Requisitos no funcionales

Los requisitos no funcionales son restricciones de los servicios o funciones que ofrece el sistema (Ej. cotas de tiempo, proceso de desarrollo, rendimiento, etc.)

Ejemplo 1. La biblioteca Central debe ser capaz de atender simultáneamente a todas las bibliotecas de la Universidad

Ejemplo 2. El tiempo de respuesta a una consulta remota no debe ser superior a 1/2 s

A su vez, hay **tres tipos** de requisitos no funcionales:

- Requisitos del producto. Especifican el comportamiento del producto (Ej. prestaciones, memoria, tasa de fallos, etc.)
- Requisitos organizativos. Se derivan de las políticas y procedimientos de las organizaciones de los clientes y

- desarrolladores (Ej. estándares de proceso, lenguajes de programación, etc.)
- Requisitos externos. Se derivan de factores externos al sistema y al proceso de desarrollo (Ej. requisitos legislativos, éticos, etc.)
- 3 - Requisitos del dominio.

Los requisitos del dominio se derivan del dominio de la aplicación y reflejan características de dicho dominio.

Pueden ser funcionales o no funcionales.

Ej. El sistema de biblioteca de la Universidad debe ser capaz de exportar datos mediante el Lenguaje de Intercomunicación de Bibliotecas de España (LIBE).

Ej. El sistema de biblioteca no podrá acceder a bibliotecas con material censurado.

Diseño del Sistema

Codificación del software

Durante esta la etapa se realizan las tareas que comúnmente se conocen como programación; que consiste, esencialmente, en llevar a código fuente, en el lenguaje de programación elegido, todo lo diseñado en la fase anterior. Esta tarea la realiza el programador, siguiendo por completo los lineamientos impuestos en el diseño y en consideración siempre a los requisitos funcionales y no funcionales (ERS) especificados en la primera etapa.

Es común pensar que la etapa de programación o codificación (algunos la llaman implementación) es la que insume la mayor parte del trabajo de desarrollo del software; sin embargo, esto puede ser relativo (y generalmente aplicable a sistemas de pequeño porte) ya que las etapas previas son cruciales, críticas y pueden llevar bastante más tiempo. Se suele hacer estimaciones de un 30% del tiempo total insumido en la programación, pero esta cifra no es consistente ya que depende en gran medida de las características del sistema, su criticidad y el lenguaje de programación elegido. En tanto menor es el nivel del lenguaje mayor será el tiempo de programación requerido, así por ejemplo se tardaría más tiempo en codificar un algoritmo en Assembly que el mismo programado en lenguaje C.

Mientras se programa la aplicación, sistema, o software en general, se realizan también tareas de depuración, esto es la labor de ir liberando al código de los errores factibles de ser hallados en esta fase (de semántica, sintáctica y lógica). Hay una suerte de solapamiento con la fase siguiente, ya que para depurar la lógica es necesario realizar

pruebas unitarias, normalmente con datos de prueba; claro es que no todos los errores serán encontrados sólo en la etapa de programación, habrán otros que se encontrarán durante las etapas subsiguientes. La aparición de algún error funcional (mala respuesta a los requerimientos) eventualmente puede llevar a retornar a la fase de diseño antes de continuar la codificación.

Durante la fase de programación, el código puede adoptar varios estados, dependiendo de la forma de trabajo y del lenguaje elegido, a saber:

- Código fuente: es el escrito directamente por los programadores en editores de texto, lo cual genera el programa. Contiene el conjunto de instrucciones codificadas en algún lenguaje de alto nivel. Puede estar distribuido en paquetes, procedimientos, librerías fuente, etc.
- Código objeto: es el código binario o intermedio resultante de procesar con un compilador el código fuente. Consiste en una **traducción completa** y de una sola vez de éste último. El código objeto no es inteligible por el ser humano (normalmente es formato binario) pero tampoco es directamente ejecutable por la computadora. Se trata de una representación intermedia entre el código fuente y el código ejecutable, a los fines de un enlace final con las rutinas de librería y entre procedimientos o bien para su uso con un pequeño intérprete intermedio [a modo de distintos ejemplos véase EUPHORIA, (intérprete intermedio), FORTRAN (compilador puro) *MSIL (Microsoft Intermediate Language)* (intérprete) y BASIC (intérprete puro, intérprete intermedio, compilador intermedio o compilador puro, depende de la versión utilizada)].
 - El código objeto **no existe** si el programador trabaja con un lenguaje **a modo de intérprete puro**, en este caso el mismo intérprete se encarga de traducir y ejecutar línea por línea el código fuente (de acuerdo al flujo del programa), en tiempo de ejecución. En este caso **tampoco existe** el o los archivos de código ejecutable. Una desventaja de esta modalidad es que la ejecución del programa o sistema es un poco más lenta que si se hiciera con un intérprete intermedio, y bastante más lenta que si existe el o los archivos de código ejecutable. Es decir no favorece el rendimiento en velocidad de ejecución. Pero una gran ventaja de la modalidad intérprete puro, es que el esta forma de trabajo facilita enormemente la tarea de depuración del código fuente (frente a la alternativa de hacerlo con un compilador puro). Frecuentemente se suele usar una forma mixta de trabajo (si el lenguaje de programación elegido lo permite), es decir inicialmente trabajar a modo de intérprete puro, y una vez

depurado el código fuente (liberado de errores) se utiliza un compilador del mismo lenguaje para obtener el código ejecutable completo, con lo cual se agiliza la depuración y la velocidad de ejecución se optimiza.

- **Código ejecutable:** Es el código binario resultado de enlazar uno o más fragmentos de código objeto con las rutinas y librerías necesarias. Constituye uno o más archivos binarios con un formato tal que el sistema operativo es capaz de cargarlo en la memoria RAM (eventualmente también parte en una memoria virtual), y proceder a su ejecución directa. Por lo anterior se dice que el código ejecutable es directamente "inteligible por la computadora". El código ejecutable, también conocido como código máquina, no existe si se programa con modalidad de "intérprete puro".

Pruebas (unitarias y de integración)

Entre las diversas pruebas de software se encuentran:

- **Prueba unitaria:** Consiste en probar o testear piezas de software elementales. Dichas pruebas se utilizan para asegurar el correcto funcionamiento de secciones de código reducidas.
- **Pruebas de integración:** Se realizan una vez que las pruebas unitarias fueron concluidas; con estas se intenta asegurar que el sistema o los subsistemas que componen las piezas individuales del software funcionen correctamente al operar en conjunto.

Instalación y paso a Producción

La instalación del software es el proceso por el cual los programas desarrollados son transferidos apropiadamente al computador destino, inicializados, y, eventualmente, configurados; todo ello con el propósito de ser ya utilizados por el usuario final. Constituye la etapa final en el desarrollo propiamente dicho del software. Luego de ésta el producto entrará en la fase de funcionamiento y producción, para el que fuera diseñado.

La instalación, dependiendo del sistema desarrollado, puede consistir en una simple copia al disco rígido destino (casos raros actualmente); o bien, más comúnmente, con una de complejidad intermedia en la que los distintos archivos componentes del software (ejecutables, librerías, datos propios, etc.) son descomprimidos y copiados a lugares específicos preestablecidos del disco; incluso se crean vínculos con otros productos, además del propio sistema operativo. Este último caso, comúnmente es un proceso bastante automático que es creado y guiado con herramientas software específicas (empaquetado y distribución, instaladores).

En productos de mayor complejidad, la segunda alternativa es la utilizada, pero es realizada y/o guiada por especialistas; puede incluso requerirse la instalación en varios y distintos computadores (instalación distribuida).

También, en software de mediana y alta complejidad normalmente es requerido un proceso de configuración y chequeo, por el cual se asignan adecuados parámetros de funcionamiento y se prueba la operatividad funcional del producto.

En productos de venta masiva las instalaciones completas, si son relativamente simples, suelen ser realizadas por los propios usuarios finales (tales como sistemas operativos, paquetes de oficina, utilitarios, etc.) con herramientas propias de instalación guiada; incluso la configuración suele ser automática. En productos de diseño específico o "a medida" la instalación queda restringida, normalmente, a personas especialistas involucradas en el desarrollo del software en cuestión.

Una vez realizada exitosamente la instalación del software, el mismo pasa a la fase de producción (operatividad), durante la cual cumple las funciones para las que fue desarrollado, es decir, es finalmente utilizado por el (o los) usuario final, produciendo los resultados esperados.

2.2 UML

UML [UML] es un lenguaje para especificar, construir, visualizar y documentar los artefactos de un sistema de software orientado a objetos (OO). Un artefacto es una información que es utilizada o producida mediante un proceso de desarrollo de software.

UML se quiere convertir en un lenguaje estándar con el que sea posible modelar todos los componentes del proceso de desarrollo de aplicaciones. Sin embargo, hay que tener en cuenta un aspecto importante del modelo: no pretende definir un modelo estándar de desarrollo, sino únicamente un lenguaje de modelado.

Otros métodos de modelaje como OMT (Object Modeling Technique) o Booch sí definen procesos concretos. En UML los procesos de desarrollo son diferentes según los distintos dominios de trabajo; no puede ser el mismo el proceso para crear una aplicación en tiempo real, que el proceso de desarrollo de una aplicación orientada a gestión, por poner un ejemplo.

Las diferencias son muy marcadas y afectan a todas las faces del proceso. El método del UML recomienda utilizar los procesos que otras metodologías tienen definidos.

A partir del año 1994, Grady Booch [Booch96] (precursor de Booch '93) y Jim Rumbaugh (creador de OMT) se unen en una empresa común, Rational Software Corporation, y comienzan a unificar sus dos métodos. Un año más tarde, en octubre de 1995, aparece UML (Unified Modeling Language) 0.8, la que se considera como la primera versión del UML. A finales de ese mismo año, Ivan Jacobson, creador de OOSE (Object Oriented Software Engineer) se añade al grupo.

Como objetivos principales de la consecución de un nuevo método que aunara los mejores aspectos de sus predecesores, sus protagonistas se propusieron lo siguiente:

- El método debía ser capaz de modelar no sólo sistemas de software sino otro tipo de sistemas reales de la empresa, siempre utilizando los conceptos de la orientación a objetos (OO).
- Crear un lenguaje para modelado utilizable a la vez por máquinas y por personas.
- Establecer un acoplamiento explícito de los conceptos y los artefactos ejecutables.
- Manejar los problemas típicos de los sistemas complejos de misión crítica.

Lo que se intenta es lograr con esto que los lenguajes que se aplican siguiendo los métodos más utilizados sigan evolucionando en conjunto y no por separado. Y además, unificar las perspectivas entre diferentes tipos de sistemas (no sólo software, sino también en el ámbito de los negocios), al aclarar las fases de desarrollo, los requerimientos de análisis, el diseño, la implementación y los conceptos internos de la OO.

Modelado de objetos

En la especificación del UML podemos comprobar que una de las partes que lo componen es un metamodelo formal. Un metamodelo es un modelo que define el lenguaje para expresar otros modelos. Un modelo en OO es una abstracción cerrada semánticamente de un sistema y un sistema es una colección de unidades conectadas que son organizadas

para realizar un propósito específico. Un sistema puede ser descrito por uno o más modelos, posiblemente desde distintos puntos de vista.

Una parte del UML define, entonces, una abstracción con significado de un lenguaje para expresar otros modelos (es decir, otras abstracciones de un sistema, o conjunto de unidades conectadas que se organizan para conseguir un propósito).

Lo que en principio puede parecer complicado no lo es tanto si pensamos que uno de los objetivos del UML es llegar a convertirse en una manera de definir modelos, no sólo establecer una forma de modelo, de esta forma simplemente estamos diciendo que UML, además, define un lenguaje con el que podemos abstraer cualquier tipo de modelo.

El UML es una técnica de modelado de objetos y como tal supone una abstracción de un sistema para llegar a construirlo en términos concretos. El modelado no es más que la construcción de un modelo a partir de una especificación.

Un modelo es una abstracción de algo, que se elabora para comprender ese algo antes de construirlo. El modelo omite detalles que no resultan esenciales para la comprensión del original y por lo tanto facilita dicha comprensión.

Los modelos se utilizan en muchas actividades de la vida humana: antes de construir una casa el arquitecto utiliza un plano, los músicos representan la música en forma de notas musicales, los artistas pintan sobre el lienzo con carboncillos antes de empezar a utilizar los óleos, etc.

Unos y otros abstraen una realidad compleja sobre unos bocetos, modelos al fin y al cabo. La OMT, por ejemplo, intenta abstraer la realidad utilizando tres clases de modelos OO: el modelo de objetos, que describe la estructura estática; el modelo dinámico, con el que describe las relaciones temporales entre objetos; y el modelo funcional que describe las relaciones funcionales entre valores. Mediante estas tres fases de construcción de modelos, se consigue una abstracción de la realidad que tiene en sí misma información sobre las principales características de ésta.

Los modelos además, al no ser una representación que incluya todos los detalles de los originales, permiten probar más fácilmente los sistemas que modelan y determinar los errores. Según se indica en la Metodología

OMT (Rumbaugh), los modelos permiten una mejor comunicación con el cliente por distintas razones:

- Es posible enseñar al cliente una posible aproximación de lo que será el producto final.
- Proporcionan una primera aproximación al problema que permite visualizar cómo quedará el resultado.
- Reducen la complejidad del original en subconjuntos que son fácilmente tratables por separado.

Se consigue un modelo completo de la realidad cuando el modelo captura los aspectos importantes del problema y omite el resto. Los lenguajes de programación que estamos acostumbrados a utilizar no son adecuados para realizar modelos completos de sistemas reales porque necesitan una especificación total con detalles que no son importantes para el algoritmo que están implementando.

En OMT se modela un sistema desde tres puntos de vista diferentes donde cada uno representa una parte del sistema y una unión lo describe de forma completa. En esta técnica de modelado se utilizó una aproximación al proceso de implementación de software habitual donde se utilizan estructuras de datos (modelo de objetos), las operaciones que se realizan con ellos tienen una secuencia en el tiempo (modelo dinámico) y se realiza una transformación sobre sus valores (modelo funcional).

UML utiliza parte de este planteamiento obteniendo distintos puntos de vista de la realidad que modela mediante los distintos tipos de diagramas que posee. Con la creación del UML se persigue obtener un lenguaje que sea capaz de abstraer cualquier tipo de sistema, sea informático o no, mediante los diagramas, es decir, mediante representaciones gráficas que contienen toda la información relevante del sistema.

Un diagrama es una representación gráfica de una colección de elementos del modelo, que habitualmente toma forma de grafo donde los arcos que conectan sus vértices son las relaciones entre los objetos y los vértices se corresponden con los elementos del modelo. Los distintos puntos de vista de un sistema real que se quieren representar para obtener el modelo se dibuja de forma que se resalten los detalles necesarios para entender el sistema.

Artefactos para el Desarrollo de Proyectos

Un artefacto es una información que es utilizada o producida mediante un proceso de desarrollo de software. Pueden ser artefactos un modelo, una descripción o un software. Los artefactos de UML se especifican en forma de diagramas, éstos, junto con la documentación sobre el sistema constituyen los artefactos principales que el modelador puede observar.

Se necesita más de un punto de vista para llegar a representar un sistema. UML utiliza los diagramas gráficos para obtener estos distintos puntos de vista de un sistema:

- Diagramas de Implementación.
- Diagramas de Comportamiento o Interacción.
- Diagramas de Casos de uso.
- Diagramas de Clases.

Ejemplo de algunos de los diagramas que utiliza UML.

Diagramas de Implementación

Se derivan de los diagramas de proceso y módulos de la metodología de Booch, aunque presentan algunas modificaciones. Los diagramas de implementación muestran los aspectos físicos del sistema. Incluyen la estructura del código fuente y la implementación, en tiempo de implementación. Existen dos tipos:

- Diagramas de componentes
- Diagrama de plataformas despliegue

4. Diagramas de componentes

Muestra la dependencia entre los distintos componentes de software, incluyendo componentes de código fuente, binario y ejecutable. Un componente es un fragmento de código software (un fuente, binario o ejecutable) que se utiliza para mostrar dependencias en tiempo de compilación.

Diagrama de plataformas o despliegue

Muestra la configuración de los componentes hardware, los procesos, los elementos de procesamiento en tiempo de ejecución y los objetos que existen en tiempo de ejecución. En este tipo de diagramas intervienen nodos, asociaciones de comunicación, componentes dentro de los nodos y objetos que se encuentran a su vez dentro de los componentes.

Un nodo es un objeto físico en tiempo de ejecución, es decir una máquina que se compone habitualmente de, por lo menos, memoria y capacidad de procesamiento, a su vez puede estar formado por otros componentes.

Diagramas de Interacción o Comportamiento

Muestran las interacciones entre objetos ocurridas en un escenario (parte) del sistema. Hay varios tipos:

Diagrama de secuencia.

Diagrama de colaboración.

Diagrama de estado.

Diagrama de actividad.

Diagrama de secuencia

Muestran las interacciones entre un conjunto de objetos, ordenadas según el tiempo en que tienen lugar. En los diagramas de este tipo intervienen objetos, que tienen un significado parecido al de los objetos representados en los diagramas de colaboración, es decir son instancias concretas de una clase que participa en la interacción.

El objeto puede existir sólo durante la ejecución de la interacción, se puede crear o puede ser destruido durante la ejecución de la interacción. Un diagrama de secuencia representa una forma de indicar el período durante el que un objeto está desarrollando una acción directamente o a través de un procedimiento.

En este tipo de diagramas también intervienen los mensajes, que son la forma en que se comunican los objetos: el objeto origen solicita (llama a) una operación del objeto destino. Existen distintos tipos de mensajes según cómo se producen en el tiempo: simples, síncronos, y asíncronos.

Los diagramas de secuencia permiten indicar cuál es el momento en el que se envía o se completa un mensaje mediante el tiempo de transición, que se especifica en el diagrama.

Diagrama de colaboración

Muestra la interacción entre varios objetos y los enlaces que existen entre ellos. Representa las interacciones entre objetos organizadas alrededor de los objetos y sus vinculaciones. A diferencia de un diagrama de secuencias, un diagrama de colaboraciones muestra las relaciones entre los objetos, no la secuencia en el tiempo en que se producen los mensajes. Los diagramas de secuencias y los diagramas de colaboraciones expresan información similar, pero en una forma diferente.

Formando parte de los diagramas de colaboración nos encontramos con objetos, enlaces y mensajes. Un objeto es una instancia de una clase que participa como una interacción, existen objetos simples y complejos. Un objeto es activo si posee un thread o hilo de control y es capaz de iniciar la actividad de control, mientras que un objeto es pasivo si mantiene datos pero no inicia la actividad.

Un enlace es una instancia de una asociación que conecta dos objetos de un diagrama de colaboración. El enlace puede ser reflexivo si conecta a un elemento consigo mismo. La existencia de un enlace entre dos objetos indica que puede existir un intercambio de mensajes entre los objetos conectados.

Los diagramas de interacción indican el flujo de mensajes entre elementos del modelo, el flujo de mensajes representa el envío de un mensaje desde un objeto a otro si entre ellos existe un enlace. Los mensajes que se envían entre objetos pueden ser de distintos tipos, también según como se producen en el tiempo; existen mensajes simples, sincrónicos, balking, timeout y asíncronos. Durante la ejecución de un diagrama de colaboración se crean y destruyen objetos y enlaces.

Diagramas de actividad

Son similares a los diagramas de flujo de otras metodologías OO. En realidad se corresponden con un caso especial de los diagramas de estado donde los estados son estados de acción (estados con una acción interna y una o más transiciones que suceden al finalizar esta acción, o lo que es lo mismo, un paso en la ejecución de lo que será un procedimiento) y las transiciones vienen provocadas por la finalización de las acciones que tienen lugar en los estados de origen.

Siempre van unidos a una clase o a la implementación de un caso de uso o de un método (que tiene el mismo significado que en cualquier otra metodología OO). Los diagramas de actividad se utilizan para mostrar el flujo de operaciones que se desencadenan en un procedimiento interno del sistema.

Diagramas de estado

Representan la secuencia de estados por los que un objeto o una interacción entre objetos pasa durante su tiempo de vida en respuesta a estímulos (eventos) recibidos. Representa lo que podemos denominar en conjunto una máquina de estados. Un estado en UML es cuando un objeto o una interacción satisface una condición, desarrolla alguna acción o se encuentra esperando un evento.

Cuando un objeto o una interacción pasa de un estado a otro por la ocurrencia de un evento se dice que ha sufrido una transición, existen varios tipos de transiciones entre objetos: simples (normales y reflexivas) y complejas.

Además una transición puede ser interna si el estado del que parte el objeto o interacción es el mismo que al que llega, no se provoca un cambio de estado y se representan dentro del estado, no de la transición. Como en todas las metodologías OO se envían mensajes, en este caso es la acción de la que puede enviar mensajes a uno o varios objetos destino

Diagramas de Casos de Uso

Unos casos de uso es una secuencia de transacciones que son desarrolladas por un sistema en respuesta a un evento que inicia un actor sobre el propio sistema. Los diagramas de casos de uso sirven para especificar la funcionalidad y el comportamiento de un sistema mediante su interacción con los usuarios y/o otros sistemas. O lo que es igual, un diagrama que muestra la relación entre los actores y los casos de uso en un sistema. Una relación es una conexión entre los elementos del modelo, por ejemplo la relación y la generalización son relaciones.

Los diagramas de casos de uso se utilizan para ilustrar los requerimientos del sistema al mostrar como reacciona una respuesta a eventos que se producen en el mismo. En este tipo de diagrama intervienen algunos conceptos nuevos: un actor es una entidad externa al sistema que se modela y que puede interactuar con él; un ejemplo de actor podría ser un usuario o cualquier otro sistema. Las relaciones entre casos de uso y actores pueden ser las siguientes:

- Un actor se comunica con un caso de uso.
- Un caso de uso extiende otro caso de uso.
- Un caso de uso usa otro caso de uso

Diagramas de Clases

Los diagramas de clases representan un conjunto de elementos del modelo que son estáticos, como las clases y los tipos, sus contenidos y las relaciones que se establecen entre ellos.

Algunos de los elementos que se pueden clasificar como estáticos son los siguientes:

Paquete: Es el mecanismo de que dispone UML para organizar sus elementos en grupos, se representa un grupo de elementos del modelo. Un sistema es un único paquete que contiene el resto del sistema, por lo tanto, un paquete debe poder anidarse, permitiéndose que un paquete contenga otro paquete.

Clases: Una clase representa un conjunto de objetos que tienen una estructura, un comportamiento y unas relaciones con propiedades parecidas. Describe un conjunto de objetos que comparte los mismos

atributos, operaciones, métodos, relaciones y significado. En UML una clase es una implementación de un tipo. Los componentes de una clase son:

- **Atributo.** Se corresponde con las propiedades de una clase o un tipo. Se identifica mediante un nombre. Existen atributos simples y complejos.
- **Operación.** También conocido como método, es un servicio proporcionado por la clase que puede ser solicitado por otras clases y que produce un comportamiento en ellas cuando se realiza.

Las clases pueden tener varios parámetros formales, son las clases denominadas plantillas. Sus atributos y operaciones vendrán definidas según sus parámetros formales. Las plantillas pueden tener especificados los valores reales para los parámetros formales, entonces reciben el nombre de clase parametrizada instanciada. Se puede usar en cualquier lugar en el que se podría aparecer su plantilla.

Relacionando con las clases nos encontramos con el término utilidad, que se corresponde con una agrupación de variables y procedimientos globales en forma de declaración de clase, también puede definirse como un estereotipo (o nueva clase generada a partir de otra ya existente) de un tipo que agrupa variables globales y procedimientos en una declaración de clase.

Los atributos y operaciones que se agrupan en una utilidad se convierten en variables y operaciones globales. Una utilidad no es fundamental para el modelado, pero puede ser conveniente durante la programación.

Metaclase: Es una clase cuyas instancias son clases. Sirven como depósito para mantener las variables de clase y proporcionan operaciones (método de clase) para inicializar estas variables. Se utilizan para construir metamodelos (modelos que se utilizan para definir otros modelos).

Tipos: Es un descriptor de objetos que tiene un estado abstracto y especificaciones de operaciones pero no su implementación. Un tipo establece una especificación de comportamiento para las clases.

Interfaz: Representa el uso de un tipo para describir el comportamiento visible externamente de cualquier elemento del modelo.

Relación entre clases: Las clases se relacionan entre sí de distintas formas, que marcan los tipos de relaciones existentes:

Asociación:

Es una relación que describe un conjunto de vínculos entre clases. Pueden ser binarias o n-arias, según se implican a dos clases o más.

Las relaciones de asociación vienen identificadas por los roles, que son los nombres que indican el comportamiento que tienen los tipos o las clases, en el caso del rol de asociación (existen otros tipos de roles según la relación a la que identifiquen). Indican la información más importante de las asociaciones.

Es posible indicar el número de instancias de una clase que participan en una relación mediante la llamada multiplicidad. Cuando la multiplicidad de un rol es mayor que 1, el conjunto de elementos que se relacionan puede estar ordenado.

Las relaciones de asociación permiten especificar qué objetos van a estar asociados con otro objeto mediante un calificador. El calificador es un atributo o conjunto de atributos de una asociación que determina los valores que indican cuales son los valores que se asociarán.

Una asociación se dirige desde una clase a otra (o un objeto a otro), el concepto de navegabilidad se refiere al sentido en el que se recorre la asociación.

Existe una forma especial de asociación, la agregación, que especifica una relación entre las clases donde el llamado "agregado" indica él todo y el "componente" es una parte del mismo.

Composición:

Es un tipo de agregación donde la relación de posesión es tan fuerte como para marcar otro tipo de relación. Las clases en UML tienen un tiempo de vida determinado, en las relaciones de composición, el tiempo de vida de la clase que es parte del todo (o agregado) viene determinado por el tiempo de vida de la clase que representa el todo, por tanto es equivalente a un atributo, aunque no lo es porque es una clase y puede funcionar como tal en otros casos.

Generalización:

Cuando se establece una relación de este tipo entre dos clases, una es una Superclase y la otra es una Subclase. La subclase comparte la estructura y el comportamiento de la superclase. Puede haber más de una clase que se comporte como subclase.

Dependencia:

Una relación de dependencia se establece entre clases (u objetos) cuando un cambio en el elemento independiente del modelo puede requerir un cambio en el elemento dependiente.

Relación de Refinamiento

Es una relación entre dos elementos donde uno de ellos especifica de forma completa al otro que ya ha sido especificado con cierto detalle.

Nuevas características del UML

Además de los conceptos extraídos de métodos anteriores, se han añadido otros nuevos que vienen a suplir carencias antiguas de la metodología de modelado. Estos nuevos conceptos son los siguientes:

- Definición de estereotipos: un estereotipo es una nueva clase de elemento de modelado que debe basarse en ciertas clases ya existentes en el metamodelo y constituye un mecanismo de extensión del modelo.
- Responsabilidades.
- Mecanismos de extensibilidad: estereotipos, valores etiquetados y restricciones.
- Tareas y procesos.
- Distribución y concurrencia (para modelar por ejemplo ActiveX/DCOM y CORBA).
- Patrones/Colaboraciones.
- Diagramas de actividad (para reingeniería de proceso de negocios)
- Clara separación de tipo, clase e instancia.
- Refinamiento (para manejar relaciones entre niveles de abstracción).
- Interfaces y componentes.

El Proceso de Desarrollo

UML no define un proceso concreto que determine las fases de desarrollo de un sistema, las empresas pueden utilizar UML como el lenguaje para definir sus propios procesos y lo único que tendrán en común con otras organizaciones que utilicen UML serán los tipos de diagramas.

UML es un método independiente del proceso. Los procesos de desarrollo deben ser definidos dentro del contexto donde se van a implementar los sistemas.

Herramientas CASE

Rational Rose es la herramienta CASE que comercializan los desarrolladores de UML y que soporta de forma completa la especificación del UML 1.1.

Esta herramienta propone la utilización de cuatro tipos de modelo para realizar un diseño del sistema, utilizando una vista estática y otra dinámica de los modelos del sistema, uno lógico y otro físico. Permite crear y refinar estas vistas creando de esta forma un modelo completo que representa el dominio del problema y el sistema de software.

Desarrollo Iterativo

Rational Rose utiliza un proceso de desarrollo iterativo controlado (controlled iterative process development), donde el desarrollo se lleva a cabo en una secuencia de iteraciones.

Cada iteración comienza con una primera aproximación del análisis, diseño e implementación para identificar los riesgos del diseño, los cuales se utilizan para conducir la iteración, primero se identifican los riesgos y después se prueba la aplicación para que éstos se hagan mínimos.

Cuando la implementación pasa todas las pruebas que se determinan en el proceso, ésta se revisa y se añaden los elementos modificados al modelo de análisis y diseño. Una vez que la actualización del modelo se ha modificado, se realiza la siguiente iteración.

Trabajo en Grupo

Rose permite que haya varias personas trabajando a la vez en el proceso iterativo controlado, para ello posibilita que cada desarrollador opere en

un espacio de trabajo privado que contiene el modelo completo y tenga un control exclusivo sobre la propagación de los cambios en ese espacio de trabajo.

También es posible descomponer el modelo en unidades controladas e integrarlas con un sistema para realizar el control de proyectos que permite mantener la integridad de dichas unidades.

Generador de Código

Se puede generar código en distintos lenguajes de programación a partir de un diseño en UML.

Ingeniería Inversa

Rational Rose proporciona mecanismos para realizar la denominada Ingeniería Inversa, es decir, a partir del código de un programa, se puede obtener información sobre su diseño.

2.3 RUP

El **Proceso Unificado Racional** (*Rational Unified Process* en inglés, habitualmente resumido como **RUP**) es un proceso de desarrollo de software y junto con el Lenguaje Unificado de Modelado UML, constituye la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas orientados a objetos.

El RUP no es un sistema con pasos firmemente establecidos, sino un conjunto de metodologías adaptables al contexto y necesidades de cada organización.

También se conoce por este nombre al software desarrollado por Rational, hoy propiedad de IBM, el cual incluye información entrelazada de diversos artefactos y descripciones de las diversas actividades. Está incluido en el **Rational Method Composer** (RMC), que permite la personalización de acuerdo a necesidades.

Originalmente se diseñó un proceso genérico y de dominio público, el Proceso Unificado, y una especificación más detallada, el ***Rational Unified Process***, que se vendiera como producto independiente.

Principios de desarrollo

El RUP está basado en 6 principios clave que son:

Adaptar el proceso

El proceso deberá adaptarse a las características propias del proyecto u organización. El tamaño del mismo, así como su tipo o las regulaciones que lo condicionen, influirán en su diseño específico. También se deberá tener en cuenta el alcance del proyecto.

Equilibrar prioridades

Los requerimientos de los diversos participantes pueden ser diferentes, contradictorios o disputarse recursos limitados. *Debe encontrarse un equilibrio que satisfaga los deseos de todos.* Gracias a este equilibrio se podrán corregir desacuerdos que surjan en el futuro.

Demostrar valor iterativamente

Los proyectos se entregan, aunque sea de un modo interno, en **etapas iteradas**. En cada iteración se analiza la opinión de los inversores, la estabilidad y calidad del producto, y se refina la dirección del proyecto así como también los riesgos involucrados

Colaboración entre equipos

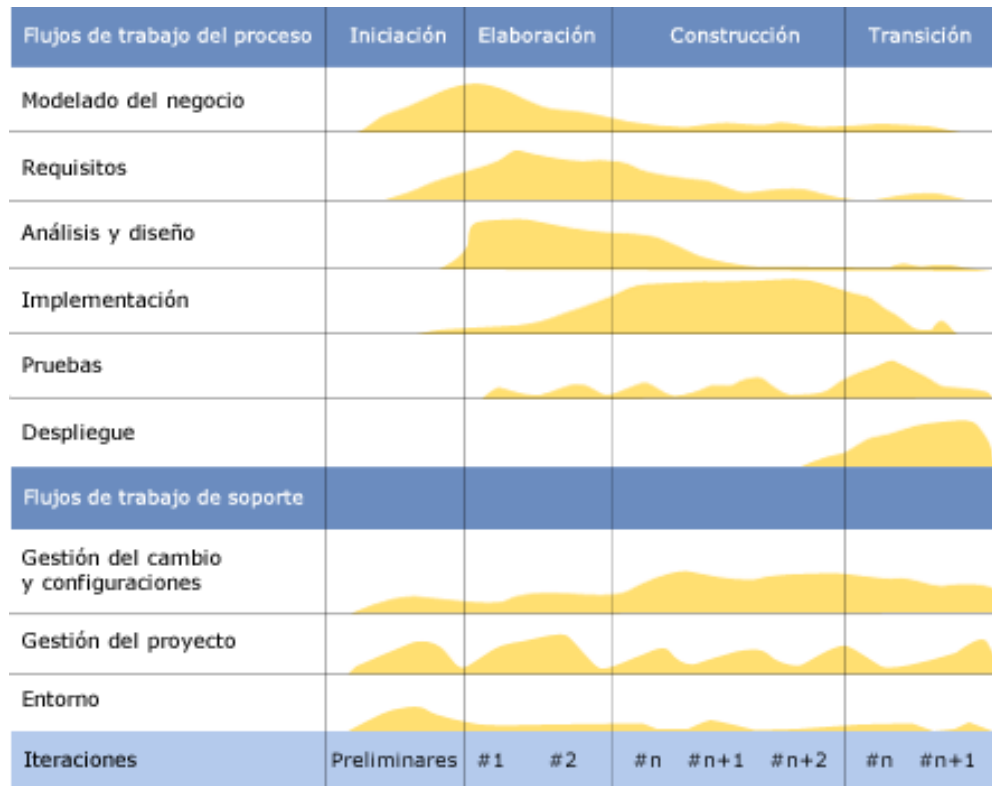
El desarrollo de software no lo hace una única persona sino múltiples equipos. Debe haber una comunicación fluida para coordinar requerimientos, desarrollo, evaluaciones, planes, resultados, etc. también esta metodología esta basada en 2 punto 1,2

Elevar el nivel de abstracción

Este principio dominante motiva el uso de conceptos reutilizables tales como patrón del software, lenguajes 4GL o marcos de referencia (frameworks) por nombrar algunos. Esto evita que los ingenieros de software vayan directamente de los requisitos a la codificación de software a la medida del cliente, sin saber con certeza qué codificar para satisfacer de la mejor manera los requerimientos y sin comenzar desde un principio pensando en la reutilización del código. Un alto nivel de abstracción también permite discusiones sobre diversos niveles y soluciones arquitectónicas. Éstas se pueden acompañar por las representaciones visuales de la arquitectura, por ejemplo con el lenguaje UML.focarse en la calidad=== El control de calidad no debe realizarse al final de cada iteración, sino en **todos** los aspectos de la producción. El

aseguramiento de la calidad forma parte del proceso de desarrollo y no de un grupo independiente.

Ciclo de vida



Esfuerzo en actividades según fase del proyecto

El ciclo de vida RUP es una implementación del Desarrollo en espiral. Fue creado ensamblando los elementos en secuencias semiordenadas. El ciclo de vida organiza las tareas en fases e iteraciones.

RUP divide el proceso en cuatro fases, dentro de las cuales se realizan varias iteraciones en número variable según el proyecto y en las que se hace un mayor o menor hincapié en las distintas actividades. En la Figura muestra cómo varía el esfuerzo asociado a las disciplinas según la fase en la que se encuentre el proyecto RUP.

Las primeras iteraciones (en las fases de Inicio y Elaboración) se enfocan hacia la comprensión del problema y la tecnología, la delimitación del ámbito del proyecto, la eliminación de los riesgos críticos, y al establecimiento de una baseline (Línea Base) de la arquitectura.

Durante la fase de inicio las iteraciones hacen mayor énfasis en actividades de modelado del negocio y de requerimientos.

En la fase de elaboración, las iteraciones se orientan al desarrollo de la baseline de la arquitectura, abarcan más los flujos de trabajo de requerimientos, modelo de negocios (refinamiento), análisis, diseño y una parte de implementación orientado a la baseline de la arquitectura.

En la fase de construcción, se lleva a cabo la construcción del producto por medio de una serie de iteraciones.

Para cada iteración se selecciona algunos Casos de Uso, se refina su análisis y diseño y se procede a su implementación y pruebas. Se realiza una pequeña cascada para cada ciclo. Se realizan tantas iteraciones hasta que se termine la implementación de la nueva versión del producto.

En la fase de transición se pretende garantizar que se tiene un producto preparado para su entrega a la comunidad de usuarios.

Como se puede observar en cada fase participan todas las disciplinas, pero que dependiendo de la fase el esfuerzo dedicado a una disciplina varía.

Principales características

- Forma disciplinada de asignar tareas y responsabilidades (quién hace qué, cuándo y cómo)
- Pretende implementar las mejores prácticas en Ingeniería de Software
- Desarrollo iterativo
- Administración de requisitos
- Uso de arquitectura basada en componentes
- Control de cambios
- Modelado visual del software
- Verificación de la calidad del software

El RUP es un producto de Rational (IBM). Se caracteriza por ser iterativo e incremental, estar centrado en la arquitectura y guiado por los casos de uso. Incluye artefactos (que son los productos tangibles del proceso como por ejemplo, el modelo de casos de uso, el código fuente, etc.) y roles (papel que desempeña una persona en un determinado momento, una persona puede desempeñar distintos roles a lo largo del proceso)....

Fases

- Establece oportunidad y alcance
- Identifica las entidades externas o actores con las que se trata
- Identifica los casos de uso

RUP comprende 2 aspectos importantes por los cuales se establecen las disciplinas:

Proceso: Las etapas de esta sección son: (Revise nuevamente la gráfica)

- Modelado de negocio
- Requisitos
- Análisis y Diseño
- Implementación
- Pruebas
- Despliegue

Soporte: En esta parte nos encontramos con las siguientes etapas:

- Gestión del cambio y configuraciones
- Gestión del proyecto
- Entorno

La estructura dinámica de RUP es la que permite que éste sea un proceso de desarrollo fundamentalmente iterativo, y en esta parte se ven inmersas las 4 fases descritas anteriormente:

- Inicio(También llamado Incepción)
- Elaboración
- Desarrollo(También llamado Implementación, Construcción)
- Cierre (También llamado Transición)

Artefactos

RUP en cada una de sus fases (pertenecientes a la estructura estática) realiza una serie de artefactos que sirven para comprender mejor tanto el análisis como el diseño del sistema (entre otros). Estos artefactos (entre otros) son los siguientes:

Inicio:

- Documento Visión
- Especificación de Requerimientos

Elaboración:

- Diagramas de caso de uso

Construcción:

- Documento Arquitectura que trabaja con las siguientes vistas:

Vista Lógica:

- Diagrama de clases
- Modelo E-R (Si el sistema así lo requiere)

Vista de Implementación:

- Diagrama de Secuencia
- Diagrama de estados
- Diagrama de Colaboración

Vista Conceptual:

- Modelo de dominio

Vista física:

- Mapa de comportamiento a nivel de hardware.

Un poco de historia

Los orígenes de RUP se remontan al modelo espiral original de Barry Boehm. Ken Hartman, uno de los contribuidores claves de RUP colaboró con Boehm en la investigación. En 1995 Rational Software compró una compañía sueca llamada Objectory AB, fundada por Ivar Jacobson, famoso por haber incorporado los casos de uso a los métodos de desarrollo orientados a objetos.

El Rational Unified Process fue el resultado de una convergencia de Rational Approach y Objectory (el proceso de la empresa Objectory AB). El primer resultado de esta fusión fue el Rational Objectory Process, la primera versión de RUP, fue puesta en el mercado en 1998, siendo el arquitecto en jefe Philippe Kruchten.

Comentarios sobre Alcance del RUP

La metodología RUP es más apropiada para proyectos grandes (Aunque también pequeños), dado que requiere un equipo de trabajo capaz de administrar un proceso complejo en varias etapas. En proyectos pequeños, es posible que no se puedan cubrir los costos de dedicación del equipo de profesionales necesarios.

Comentarios sobre Metodología

Por otro lado, en lo que se refiere a la metodología esta comprende tres frases claves: Dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental.

En lo referente a dirigido por los casos de uso, está enfocado hacia el cliente y se utilizan con algunas modificaciones tal vez, hasta la disciplina de pruebas, en la cual, un caso de uso puede a su vez tener uno o más casos de prueba.

2.4 MDA

MDA es el acrónimo de Model Driven Architecture, en español Arquitectura dirigida por modelos.

Podemos definir a MDA como un marco de trabajo que ha definido el Object Management Group (OMG).

El OMG es una organización de compañías de sistemas de información creada en 1990 con el fin de potenciar el desarrollo de aplicaciones orientadas a objetos distribuidas. Esta organización ha definido estándares importantes como UML, CORBA, MOF, entre otros. Su sitio Web es <http://www.omg.org>.

En los últimos años se ha tomado mayor interés e importancia al modelado en el desarrollo de cualquier tipo de software, debido a la facilidad que ofrece un buen diseño tanto a la hora de desarrollar como al hacer la integración y mantenimiento de sistemas de software.

Es así que en el año 2001, el OMG definió un marco de trabajo nuevo llamado MDA. La clave del MDA es la importancia de los modelos en el proceso de desarrollo de software. MDA propone la definición y uso de modelos a diferente nivel de abstracción, así como la posibilidad de la generación automática de código a partir de los modelos definidos y de las reglas de transformación entre dichos modelos.

Ciclo de vida del desarrollo con MDA:

1. Modelo independiente de la plataforma (PIM)
2. Modelo específico a la plataforma (PSM)
3. Código

Usando la metodología MDA, la funcionalidad del sistema será definida en primer lugar como un modelo independiente de la plataforma (Platform-Independent Model o PIM) a través de un lenguaje específico para el dominio del que se trate.

Dado un modelo de definición de la plataforma (Platform Definition Model o PDM) correspondiente a CORBA, .NET, web, etc., el modelo PIM puede traducirse entonces a uno o más modelos específicos de la plataforma (Platform-specific models o PSMs) para la implementación

correspondiente, usando diferentes lenguajes específicos del dominio, o lenguajes de propósito general como Java, C#, Python, etc. La traducción entre el PIM y los PSMs se realizan normalmente utilizando herramientas automatizadas, como herramientas de transformación de modelos (por ejemplo aquellas herramientas que cumplen con el nuevo estándar de OMG denominado QVT).

El proceso completo se encuentra documentado en un documento que actualiza y mantiene OMG denominado la Guía MDA. Martin Sullon

Los principios de MDA pueden aplicarse a otras áreas como el modelado de procesos de negocios donde el PIM, independiente de la tecnología y de la arquitectura es adaptado tanto a los sistemas como a los procesos manuales.

El modelo MDA está relacionado con múltiples normas, incluyendo el lenguaje de modelado unificado (Unified Modeling Language o UML), Meta-Object Facility (MOF), XML Metadata Interchange (XMI), Enterprise Distributed Object Computing (EDOC), el Software Process Engineering Metamodel (SPEM) y el Common Warehouse Metamodel (CWM).

Fíjese que el término "arquitectura" en los metamodelos no se refiere a la arquitectura del sistema modelizado sino más bien a la arquitectura de los distintos estándares y formas del modelo que sirven de base tecnológica al MDA.

El Object Management Group mantiene la marca registrada sobre MDA, así como sobre varios términos similares incluyendo MDD (Model Driven Development), Model Driven Application Development, Model Based Application Development, Model Based Programming y otros similares.

El acrónimo principal que aun no ha sido depositado por OMG hasta ahora es MDE. A consecuencia de esto, el acrónimo MDE es usado actualmente por la comunidad investigadora internacional cuando se refieren a ideas relacionadas con la ingeniería de modelos sin centrarse exclusivamente en los estándares OMG.

Acercamiento MDA

Uno de los principales objetivos de MDA es separar el diseño de la arquitectura y de las tecnologías de construcción, facilitando que el diseño y la arquitectura puedan ser alterados independientemente. El diseño alberga los requerimientos funcionales (casos de uso) mientras que la arquitectura proporciona la infraestructura a través de la cual se hacen efectivos requerimientos no funcionales como la escalabilidad, fiabilidad o rendimiento.

MDA se asegura de que el modelo independiente de la plataforma (PIM), el cual representa un diseño conceptual que concreta los requerimientos funcionales, sobrevive a los cambios que se produzcan en las tecnologías de fabricación y en las arquitecturas software. De particular importancia en MDA es la noción de transformación de modelos. Uno de los estándares definidos para la transformación de modelos se denomina QVT.

Herramientas MDA

Un amplio conjunto de herramientas con soporte para MDA se están desarrollando por los principales fabricantes y proyectos Open Source. Estas herramientas permiten comúnmente la especificación rudimentaria de arquitecturas. Algunos ejemplos simples de estas especificaciones de arquitecturas incluyen:

- seleccionar una de las arquitecturas de referencia tales como Java EE o Microsoft .NET.
- Especificar la arquitectura a un nivel de mayor detalle incluyendo tecnologías de la capa de presentación, de la capa de negocio, de persistencia o tecnología de mapeo de persistencia (como p.e. mapeo relacional a objeto).

Conferencias

Entre las distintas conferencias sobre el tema podemos citar la ECMDA, Conferencia Europea sobre MDA, o incluso MoDELS, anteriormente llamadas conferencias <<UML>>. También hay distintas conferencias y seminarios de trabajo (en OOPSLA, ECOOP y demás) orientadas en aspectos más específicos de MDA como la transformación de modelos, la composición o su generación.