Contenido Temático del Tema III: **Proceso del Software (Metodologías)**

El proceso de ingeniería del software. Modelos de procesos. Modelo Lineal Secuencial (cascada). Modelo de Prototipos. Modelo Rápido de Aplicaciones (DRA). Modelos Evolutivos (incremental, espiral, victoria victoria, DUM, etc.). Modelo de desarrollo concurrente y basado en componentes. Modelos de Métodos Formales. Proceso Unificado de Desarrollo (PUDs). Modelo Metodologías ágiles (scrum, xp, crystal, fdd, dsdm, etc.). Tendencias en metodologías de desarrollo de software. Herramientas CASE-I para aplicar el desarrollo de metodologías.

El proceso de Ingeniería del Software

Un proceso del software es un conjunto de actividades que conducen a la creación de un producto software. Estas actividades pueden consistir en el desarrollo de software desde cero. Sin embargo, cada vez más, se desarrolla nuevo software ampliando y modificando los sistemas existentes y configurando e integrando software comercial o componentes del sistema.
Los procesos del software son complejos y, como todos los procesos intelectuales, y creativos, dependen de las personas que toman decisiones y juicios. Debido a la necesidad de juzgar y crear, los intentos para automatizar estos procesos han tenido un éxito limitado. Las herramientas de ingeniería del software asistida por computadora (CASE) pueden ayudar a algunas actividades del proceso, sin embargo, esta sigue siendo limitada.
Una razón por la cual la eficacia de las herramientas CASE está limitada se halla en la inmensa diversidad de procesos del software. No existe un proceso ideal, y muchas organizaciones han desarrollado su propio enfoque para el desarrollo del software. Los procesos han evolucionado para explotar las capacidades de las personas de una organización, así como las características específicas de los sistemas que se están desarrollando. Para algunos sistemas, como los sistemas críticos, se requiere un proceso de desarrollo muy estructurado. Para sistemas de negocio, con requerimientos rápidamente cambiantes, un proceso flexible y ágil probablemente sea más efectivo.
Aunque existen muchos procesos diferentes de software, algunas actividades fundamentales son comunes para todos ellos:

- *Especificación del software*. Se debe definir la funcionalidad del software y las restricciones en su operación.
- *Diseño e implementación del software*. Se debe producir software que cumpla su especificación.
- *Validación del software*. Se debe validar el software para asegurar que hace lo que el
- cliente desea.
- *Evolución del software*. El software debe evolucionar para cubrir las necesidades cambiantes del cliente.

Aunque no existe un proceso del software “ideal”, en las organizaciones existen enfoques para mejorarlos. Los procesos pueden incluir técnicas anticuadas o no aprovecharse de

las mejores prácticas en la ingeniería del software industrial. De hecho, muchas organizaciones aún no aprovechan los métodos de la ingeniería del software en el desarrollo de su software.
Los procesos del software se pueden mejorar por la estandarización del proceso donde la diversidad de los procesos del software en una organización sea reducida. Esto conduce a mejorar la comunicación y a una reducción del tiempo de formación, y hace la ayuda al proceso automatizado más económica. La estandarización también es un primer paso importante para introducir nuevos métodos, técnicas y buenas prácticas de ingeniería del software.

Modelos del proceso del software

Un modelo del proceso del software es una representación abstracta de un proceso del software. Cada modelo de proceso representa un proceso desde una perspectiva particular, y así proporciona sólo información parcial sobre ese proceso.
Estos modelos generales no son descripciones definitivas de los procesos del software. Más bien, son abstracciones de los procesos que se pueden utilizar para explicar diferentes enfoques para el desarrollo de software. Puede pensarse en ellos como marcos de trabajo (framework) del proceso que pueden ser extendidos y adaptados para crear procesos más específicos de ingeniería del software.
Algunos modelos de procesos que se incluyen en esta sección:

1. *El modelo en cascada*. Considera las actividades fundamentales del proceso de especificación, desarrollo, validación y evolución, y los representa como fases separadas del proceso, tales como la especificación de requerimientos, el diseño del software, la implementación, las pruebas, etcétera.
2. *Desarrollo evolutivo*. Este enfoque entrelaza las actividades de especificación, desarrollo y validación. Un sistema inicial se desarrolla rápidamente a partir de especificaciones abstractas. Éste se refina basándose en las peticiones del cliente para producir un sistema que satisfaga sus necesidades.
3. *Ingeniería del software basada en componentes*. Este enfoque se basa en la existencia de un número significativo de componentes reutilizables. El proceso de desarrollo del sistema se enfoca en integrar estos componentes en el sistema más que en desarrollarlos desde cero.

Estos tres modelos de procesos genéricos se utilizan ampliamente en la práctica actual de la ingeniería del software. No se excluyen mutuamente y a menudo se utilizan juntos, especialmente para el desarrollo de sistemas grandes. De hecho, el Proceso Unificado de Rational, como otros, combina elementos de todos estos modelos. Los subsistemas dentro de un sistema más grande pueden ser desarrollados utilizando enfoques diferentes. Por lo tanto, aunque es conveniente estudiar estos modelos separadamente, debe entenderse que, en la práctica, a menudo se combinan (híbridos).
Se han propuesto todo tipo de variantes de estos procesos genéricos y pueden ser usados en algunas organizaciones. La variante más importante es probablemente el desarrollo formal de sistemas, donde se crea un modelo formal matemático de un sistema. Este modelo se transforma entonces, usando transformaciones matemáticas que preservan su consistencia, en código ejecutable.

El ejemplo más conocido de un proceso de desarrollo formal es el proceso de sala limpia, el cual fue originalmente desarrollado por IBM. En el proceso de sala limpia, cada incremento del software se especifica formalmente, y esta especificación se transforma en una implementación. La corrección del software se demuestra utilizando un enfoque formal. No hay pruebas para defectos en el proceso, y las pruebas del sistema se centran en evaluar su fiabilidad.
Tanto el enfoque de sala limpia como otro enfoque para desarrollo formal basado en el método B, son particularmente apropiados para el desarrollo de sistemas que tienen estrictos requerimientos de seguridad, fiabilidad o protección. El enfoque formal simplifica la producción de un caso de seguridad o protección que demuestre a los clientes u organismos de certificación que el sistema realmente cumple los requerimientos de seguridad y protección.
Fuera de estos ámbitos especializados, los procesos basados en transformaciones formales no se utilizan en general. Requieren una pericia especializada y, en realidad, para la mayoría de los sistemas este proceso no ofrece ventajas importantes de coste o calidad sobre otros enfoques para el desarrollo de sistemas.

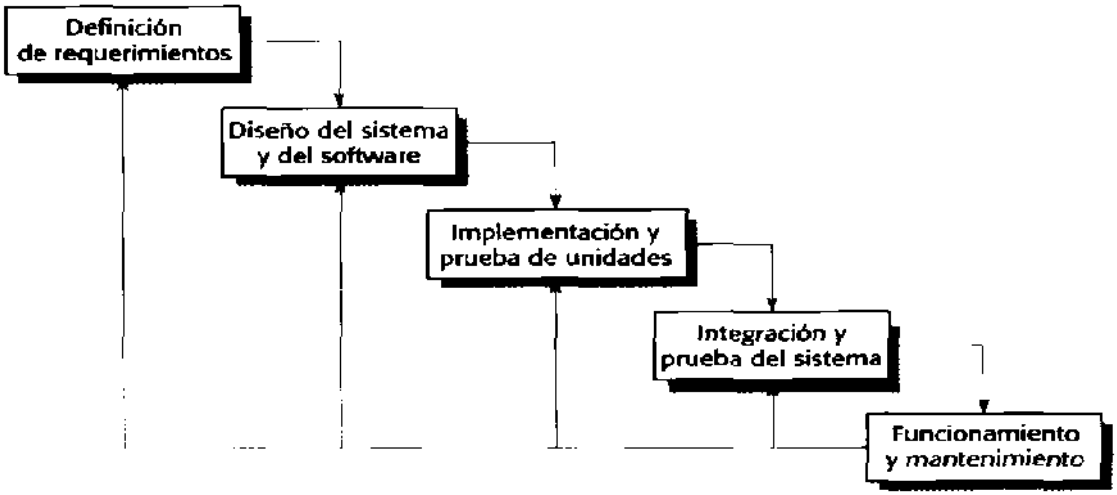
Actividades del proceso

Las tres actividades básicas del proceso de especificación, desarrollo, validación y evolución se organizan de forma distinta en diferentes procesos del desarrollo.

En el enfoque en cascada, están organizadas en secuencia, mientras que en el desarrollo evolutivo se entrelazan.
Cómo se llevan a cabo estas actividades depende del tipo de software, de las personas y de la estructura organizacional implicadas. No hay una forma correcta o incorrecta de organizar estas actividades, y el objetivo de esta sección es simplemente proporcionar una introducción de cómo se pueden organizar.

El modelo en cascada

El primer modelo de proceso de desarrollo de software que se publicó se derivó de procesos de ingeniería de sistemas más generales. Este modelo se muestra en la Figura siguiente



Debido a la cascada de una fase a otra, dicho modelo se conoce como modelo en cascada o como ciclo de vida del software. Las principales etapas de este modelo se transforman en actividades fundamentales de desarrollo:

- *Análisis y definición de requerimientos.* Los servicios, restricciones y metas del sistema se definen a partir de las consultas con los usuarios. Entonces, se definen en detalle y sirven como una especificación del sistema.
- *Diseño del sistema y del software.* El proceso de diseño del sistema divide los requerimientos en sistemas hardware o software. Establece una arquitectura completa del sistema. El diseño del software identifica y describe las abstracciones fundamentales del sistema software y sus relaciones.
- *Ímplementación y prueba de unidades.* Durante esta etapa, el diseño del software se lleva a cabo como un conjunto o unidades de programas. La prueba de unidades implica verificar que cada una cumpla su especificación.
- *Integración* y *prueba del sistema.* Los programas o las unidades individuales de programas se integran y prueban como un sistema completo para asegurar que se cumplan los requerimientos del software. Después de las pruebas, el sistema software se entrega al cliente.
- *Funcionamiento y mantenimiento.* Por lo general (aunque no necesariamente), ésta es la fase más larga del ciclo de vida, El sistema se instala y se pone en funcionamiento práctico. El mantenimiento implica corregir errores no descubiertos en las etapas anteriores del ciclo de vida, mejorar la Ímplementación de las unidades del sistema y resaltar los servicios del sistema una vez que se descubren nuevos requerimientos.

En principio, el resultado de cada fase es uno o más documentos aprobados («firmados»). La siguiente fase no debe empezar hasta que la fase previa haya finalizado. En la práctica, estas etapas se superponen y proporcionan información a las otras. Durante el diseño se identifican los problemas con los requerimientos; durante el diseño del código se encuentran problemas, y así sucesivamente. El proceso del software no es un modelo lineal simple, sino que implica una serie de iteraciones de las actividades de desarrollo.

Debido a los costos de producción y aprobación de documentos, las iteraciones son costosas e implican rehacer el trabajo. Por lo tanto, después de un número reducido de iteraciones, es normal congelar partes del desarrollo, como la especificación, y continuar con las siguientes etapas de desarrollo. Los problemas se posponen para su resolución, se pasan por alto o se programan. Este congelamiento prematuro de requerimientos puede implicar que el sistema no haga lo que los usuarios desean. También puede conducir a sistemas mal estructurados debido a que los problemas de diseño se resuelven mediante trucos de implementación.

Durante la fase final del ciclo de vida (funcionamiento y mantenimiento), el software se pone en funcionamiento. Se descubren errores y omisiones en los requerimientos originales del software. Los errores de programación y de diseño emergen y se identifica la necesidad de una nueva funcionalidad. Por tanto, el sistema debe evolucionar para mantenerse útil. Hacer estos cambios (mantenimiento del software) puede implicar repetir etapas previas del proceso.

Las ventajas del modelo en cascada son que la documentación se produce en cada fase y que éste cuadra con otros modelos del proceso de ingeniería. Su principal problema es su inflexibilidad al dividir el proyecto en distintas etapas. Se deben hacer compromisos en las etapas iníciales, lo que hace difícil responder a los cambios en los requerimientos del cliente.

Por lo tanto, el modelo en cascada sólo se debe utilizar cuando los requerimientos se comprendan bien y sea improbable que cambien radicalmente durante el desarrollo del sistema.

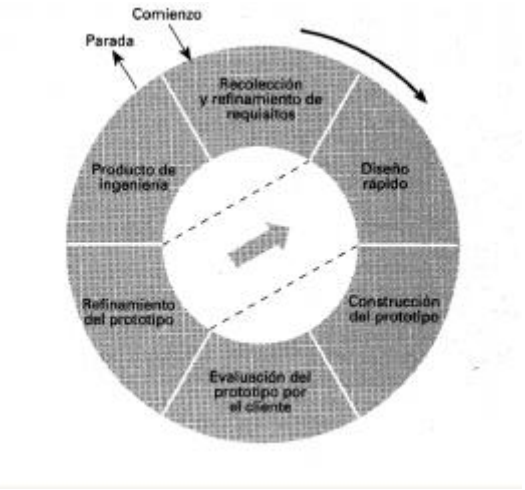
Sin embargo, el modelo refleja el tipo de modelo de proceso usado en otros proyectos de la ingeniería. Por consiguiente, los procesos del software que se basan en este enfoque se siguen utilizando para el desarrollo de software, particularmente cuando éste es parte de proyectos grandes de ingeniería de sistemas.

Modelo de Prototipos.

También conocido como desarrollo con prototipación o modelo 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.

Este modelo se utilizan para dar al usuario una vista preliminar de parte del software. Este modelo es básicamente prueba y error ya que si al usuario no le gusta una parte del prototipo significa que la prueba fallo por lo cual se debe corregir el error que se tenga hasta que el usuario quede satisfecho.

Además el prototipo debe ser construido en poco tiempo, usando los programas adecuados y no se debe utilizar mucho dinero pues a partir de que este sea aprobado nosotros podemos iniciar el verdadero desarrollo del software. Pero eso si al construir el prototipo nos asegura que nuestro software sea de mejor calidad, además de que su interfaz sea de agrado para el usuario. Un prototipo podrá ser construido solo si con el software es posible experimentar.



Etapas

- *Recolección y refinamiento de requisitos*
- *Modelado, diseño rápido*
- *Construcción del Prototipo*
- *Desarrollo, evaluación del prototipo por el cliente*
- *Refinamiento del prototipo*
- *Producto de Ingeniería*

Cómo se lleva a cabo

Se comienza elaborando un prototipo del producto final: qué aspecto tendrá, cómo funcionará. Para muchas interfaces de usuario, este modelo puede resultar tan simple como unos dibujos con lápiz y papel o tan complejo como el propio código operativo final. Cuanto más próximo se encuentre el prototipo al producto real, mejor será la evaluación, si bien se pueden obtener magníficos resultados con prototipos de baja fidelidad.

Ventajas

- No modifica el flujo del ciclo de vida
- Reduce el riesgo de construir productos que no satisfagan las necesidades de los usuarios
- Reduce costo y aumenta la probabilidad de éxito
- Exige disponer de las herramientas adecuadas
- 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.

Para que sea efectivo

- Debe ser un sistema con el que se pueda experimentar
- Debe ser comparativamente barato(menor que el 10%)
- Debe desarrollarse rapidamente

- Enfasis en la interfaz de usuario
- Equipo de desarrollo reducido
- Herramientas y lenguajes adecuadas

Desventajas

- Debido a que el usuario ve que el prototipo funciona piensa que este es el producto terminado y no entienden que recién se va a desarrollar el software.
- El desarrolador puede caer en la tentación de ampliar el prototipo para construir el sistema final sin tener en cuenta los compromisos de calidad y mantenimiento que tiene con el cliente

Tipos de Modelo de Prototipos

- Modelo de Prototipos rápido: Metodología de diseño que desarrolla rápidamente nuevos diseños, los evalúa y prescinde del prototipo cuando el próximo diseño es desarrollado mediante un nuevo prototipo.
- Modelo de Prototipos reutilizable: También conocido como "Evolutionary Prototyping"; no se pierde el esfuerzo efectuado en la construcción del prototipo pues sus partes o el conjunto pueden ser utilizados para construir el producto real. Mayormente es utilizado en el desarrollo de software, si bien determinados productos de hardware pueden hacer uso del prototipo como la base del diseño de moldes en la fabricación con plásticos o en el diseño de carrocerías de automóviles.
- Modelo de Prototipos Modular: También conocido como Prototipado Incremental (Incremental prototyping); se añaden nuevos elementos sobre el prototipo a medida que el ciclo de diseño progresa.
- Modelo de Prototipos Horizontal: El prototipo cubre un amplio número de aspectos y funciones pero la mayoría no son operativas. Resulta muy útil para evaluar el alcance del producto, pero no su uso real.
- Modelo de Prototipos Vertical: El prototipo cubre sólo un pequeño número de funciones operativas. Resulta muy útil para evaluar el uso real sobre una pequeña parte del producto.
- Modelo de Prototipos de Baja-fidelidad: El prototipo se implementa con papel y lápiz, emulando la función del producto real sin mostrar el aspecto real del mismo. Resulta muy útil para realizar tests baratos.
- Modelo de Prototipos de Alta-fidelidad: El prototipo se implementa de la forma más cercana posible al diseño real en términos de aspecto, impresiones, interacción y tiempo.

Tipos de prototipos

Hay dos clases de prototipos el desechable y el evolucionario.

- El desechable: nos sirve para eliminar dudas sobre lo que realmente quiere el cliente además para desarrollar la interfaz que más le convenga al cliente.
- El evolucionario: es un modelo parcialmente construido que puede pasar de ser prototipo a ser software pero no tiene una buena documentación y calidad.

Ingeniería del software basada en componentes (CBSE)

En la mayoría de los proyectos de software existe algo de reutilización de software. Por lo general, esto sucede informalmente cuando las personas que trabajan en el proyecto conocen diseños o código similares al requerido. Los buscan, los modifican según lo creen necesario y los incorporan en el sistema. En el enfoque evolutivo, la reutilización es a menudo indispensable para el desarrollo rápido de sistemas.
Esta reutilización informal es independiente del proceso de desarrollo que se utilice. Sin embargo, en los últimos años, ha surgido un enfoque de desarrollo de software denominado ingeniería del software basada en componentes (CBSE) que se basa en la reutilización, el cual se está utilizando de forma amplia.
Este enfoque basado en la reutilización se compone de una gran base de componentes software reutilizables y de algunos marcos de trabajo de integración para éstos. Algunas veces estos componentes son sistemas por sí mismos que se pueden utilizar para proporcionar una funcionalidad específica, como dar formato al texto o efectuar cálculos numéricos. En la Figura siguiente se muestra el modelo del proceso genérico para la CBSE.

Aunque la etapa de especificación de requerimientos y la de validación son comparables con otros procesos, las etapas intermedias en el proceso orientado a la reutilización son diferentes.
Estas etapas son:

- *Análisis de componentes.* Dada la especificación de requerimientos, se buscan los componentes para implementar esta especificación. Por lo general, no existe una concordancia exacta y los componentes que se utilizan sólo proporcionan parte de la funcionalidad requerida.
- *Modificación de requerimientos.* En esta etapa, los requerimientos se analizan utilizando información acerca de los componentes que se han descubierto. Entonces, estos componentes se modifican para reflejar los componentes disponibles. Si las modificaciones no son posibles, la actividad de análisis de componentes se puede llevar a cabo nuevamente para buscar soluciones alternativas.
- *Diseño del sistema con reutilización.* En esta fase se diseña o se reutiliza un marco de trabajo para el sistema. Los diseñadores tienen en cuenta los componentes que se reutilizan y organizan el marco de trabajo para que los satisfaga. Si los componentes reutilizables no están disponibles, se puede tener que diseñar nuevo software.

- *Desarrollo e integración.* Para crear el sistema, el software que no se puede adquirir externamente se desarrolla, y los componentes y los sistemas COTS se integran. En este modelo, la integración de sistemas es parte del proceso de desarrollo, más que una actividad separada.

La ingeniería del software basada en componentes tiene la ventaja obvia de reducir la cantidad de software a desarrollarse y así reduce los costos y los riesgos. Por lo general, también permite una entrega más rápida del software. Sin embargo, los compromisos en los requerimientos son inevitables, y esto puede dar lugar a un sistema que no cumpla las necesidades reales de los usuarios. Más aún: si las nuevas versiones de los componentes reutilizables no están bajo el control de la organización que los utiliza, se pierde parte del control sobre la evolución del sistema.
La CBSE tiene mucho en común con un enfoque que está surgiendo para el desarrollo de sistemas que se basa en la integración de servicios web de una serie de proveedores.

Iteración de procesos

Los cambios son inevitables en todos los proyectos de software grandes. Los requerimientos del sistema cambian cuando el negocio que procura el sistema responde a las presiones externas.
Las prioridades de gestión cambian. Cuando se dispone de nuevas tecnologías, cambian los diseños y la implementación. Esto significa que el proceso del software no es un proceso único; más bien, las actividades del proceso se repiten regularmente conforme el sistema se rehace en respuesta a peticiones de cambios.
Dos modelos de procesos que han sido diseñados explícitamente para apoyar la iteración de procesos:

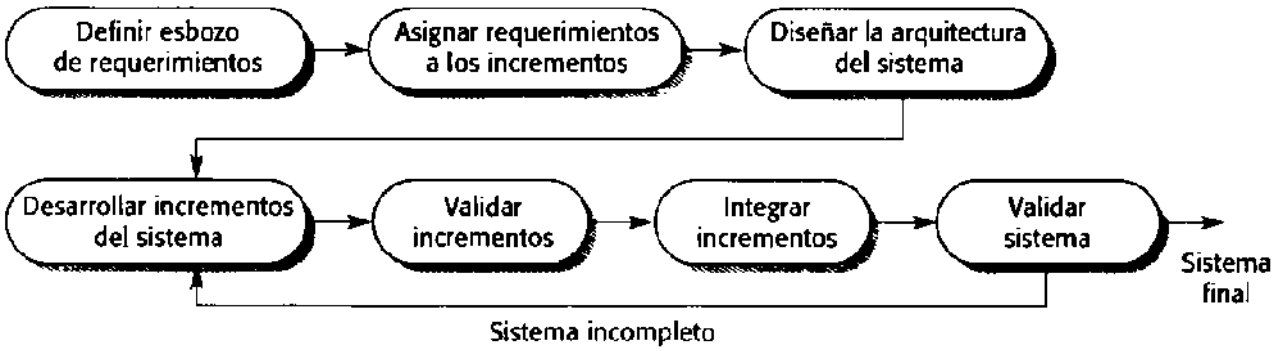
1. *Entrega incremental.* La especificación, el diseño y la implementación del software se dividen en una serie de incrementos, los cuales se desarrollan por turnos;
2. *Desarrollo en espiral* El desarrollo del sistema gira en espiral hacia fuera, empezando con un esbozo inicial y terminando con el desarrollo final del mismo.

La esencia de los procesos iterativos es que la especificación se desarrolla junto con el software.
Sin embargo, esto crea conflictos con el modelo de obtención de muchas organizaciones donde la especificación completa del sistema es parte del contrato de desarrollo del mismo.
En el enfoque incremental, no existe una especificación completa del sistema hasta que el incremento final se especifica. Esto requiere un nuevo tipo de contrato, que a los clientes grandes como las agencias del gobierno les puede ser difícil de incorporar.

Entrega incremental

El modelo de desarrollo en cascada requiere que los clientes de un sistema cumplan un conjunto de requerimientos antes de que se inicie el diseño y que el diseñador cumpla estrategias particulares de diseño antes de la implementación. Los cambios de requerimientos implican rehacer el trabajo de captura de éstos, de diseño e

implementación. Sin embargo, la separación en el diseño y la implementación deben dar lugar a sistemas bien documentados susceptibles de cambio. En contraste, un enfoque de desarrollo evolutivo permite que los requerimientos y las decisiones de diseño se retrasen, pero también origina un software que puede estar débilmente estructurado y difícil de comprender y mantener.
La entrega incremental (Figura siguiente) es un enfoque intermedio que combina las ventajas de estos modelos.



En un proceso de desarrollo incremental, los clientes identifican, a grandes rasgos, los servicios que proporcionará el sistema. Identifican qué servicios son más importantes y cuáles menos. Entonces, se definen varios incrementos en donde cada uno proporciona un subconjunto de la funcionalidad del sistema. La asignación de servicios a los incrementos depende de la prioridad del servicio con los servicios de prioridad más alta entregados primero.
Una vez que los incrementos del sistema se han identificado, los requerimientos para los servicios que se van a entregar en el primer incremento se definen en detalle, y éste se desarrolla.
Durante el desarrollo, se puede llevar a cabo un análisis adicional de requerimientos para los requerimientos posteriores, pero no se aceptan cambios en los requerimientos para el incremento actual.
Una vez que un incremento se completa y entrega, los clientes pueden ponerlo en servicio.
Esto significa que tienen una entrega temprana de parte de la funcionalidad del sistema. Pueden experimentar con el sistema, lo cual les ayuda a clarificar sus requerimientos para los incrementos posteriores y para las últimas versiones del incremento actual. Tan pronto como se completan los nuevos incrementos, se integran en los existentes de tal forma que la funcionalidad del sistema mejora con cada incremento entregado. Los servicios comunes se pueden implementar al inicio del proceso o de forma incremental tan pronto como sean requeridos por un incremento.
Este proceso de desarrollo incremental tiene varias ventajas:

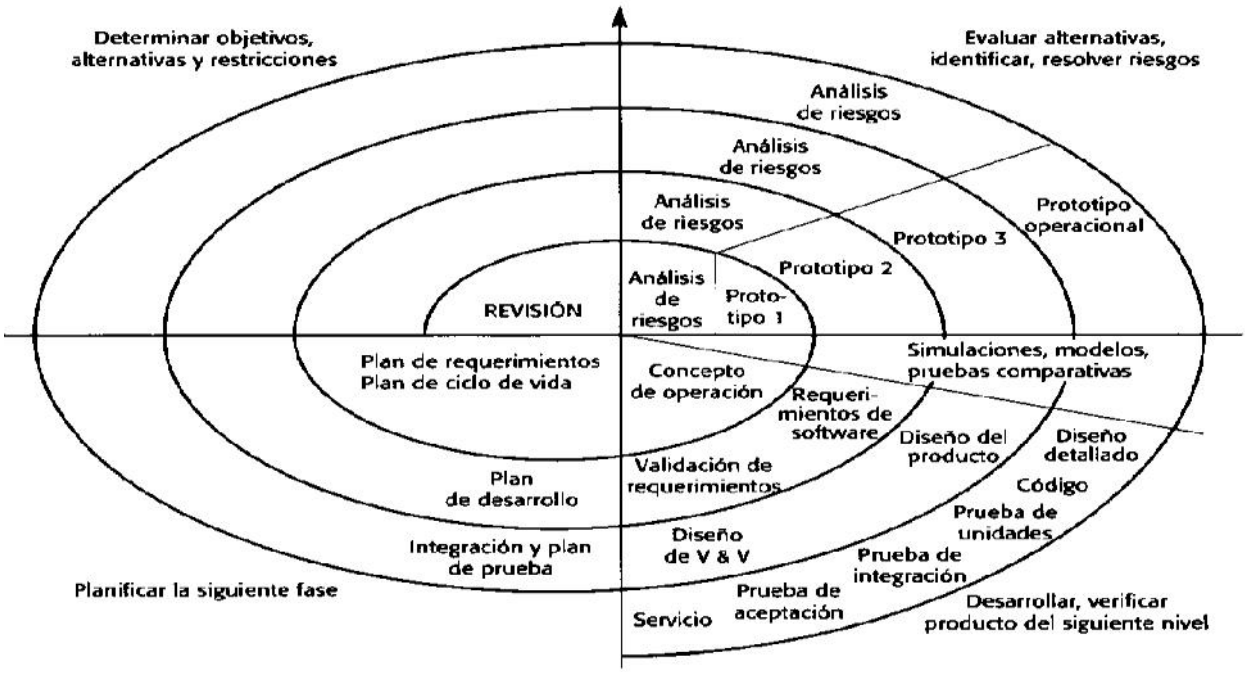
- Los clientes no tienen que esperar hasta que el sistema completo se entregue para sacar provecho de él. El primer incremento satisface los requerimientos más críticos de tal forma que pueden utilizar el software inmediatamente.
- Los clientes pueden utilizar los incrementos iniciales como prototipos y obtener experiencia sobre los requerimientos de los incrementos posteriores del sistema.

- Existe un bajo riesgo de un fallo total del proyecto. Aunque se pueden encontrar problemas en algunos incrementos, lo normal es que el sistema se entregue de forma satisfactoria al cliente.
- Puesto que los servicios de más alta prioridad se entregan primero, y los incrementos posteriores se integran en ellos, es inevitable que los servicios más importantes del sistema sean a los que se les hagan más pruebas. Esto significa que es menos probable que los clientes encuentren fallos de funcionamiento del software en las partes más importantes del sistema.

Sin embargo, existen algunos problemas en el desarrollo incremental. Los incrementos deben ser relativamente pequeños (no más de 20.000 líneas de código) y cada uno debe entregar alguna funcionalidad del sistema. Puede ser difícil adaptar los requerimientos del cliente a incrementos de tamaño apropiado. Más aún, muchos de los sistemas requieren un conjunto de recursos que se utilizan en diferentes partes del sistema. Puesto que los requerimientos no se definen en detalle hasta que un incremento se implementa, puede ser difícil identificar los recursos comunes que requieren todos los incrementos.
Se ha desarrollado una variante de este enfoque incremental denominada *programación extrema*. Ésta se basa en el desarrollo y la entrega de incrementos de funcionalidad muy pequeños, en la participación del cliente en el proceso, en la mejora constante del código y en la programación por parejas.

Desarrollo en espiral

El modelo en espiral del proceso del software (figura siguiente) fue originalmente propuesto por Boehm.



Más que representar el proceso del software como una secuencia de actividades con retrospectiva de una actividad a otra, se representa como una espiral. Cada ciclo en la espiral representa una fase del proceso del software. Así. el ciclo más interno podría referirse a la viabilidad del sistema, el siguiente ciclo a la definición de requerimientos, el siguiente ciclo al diseño del sistema, y así sucesivamente.

Cada ciclo de la espiral se divide en cuatro sectores:

- *Definición de objetivos.* Para esta fase del proyecto se definen los objetivos específicos. Se identifican las restricciones del proceso y el producto, y se traza un plan detallado de gestión. Se identifican los riesgos del proyecto. Dependiendo de estos riesgos, se planean estrategias alternativas.
- *Evaluación y reducción de riesgos.* Se lleva a cabo un análisis detallado para cada uno de los riesgos del proyecto identificados. Se definen los pasos para reducir dichos riesgo. Por ejemplo, si existe el riesgo de tener requerimientos inapropiados, se puede desarrollar un prototipo del sistema.
- *Desarrollo y validación.* Después de la evaluación de riesgos, se elige un modelo para el desarrollo del sistema. Por ejemplo, si los riesgos en la interfaz de usuario son dominantes, un modelo de desarrollo apropiado podría ser la construcción de prototipo evolutivos. Si los riesgos de seguridad son la principal consideración, un desarrollo basado en transformaciones formales podría ser el más apropiado, y así sucesivamente. El modelo en cascada puede ser el más apropiado para el desarrollo si el mayor riesgo identificado es la integración de los subsistemas.
- *Planificación.* El proyecto se revisa y se toma la decisión de si se debe continuar con un ciclo posterior de la espiral. Si se decide continuar, se desarrollan los planes para la siguiente fase del proyecto.

La diferencia principal entre el modelo en espiral y los otros modelos del proceso del software es la consideración explícita del riesgo en el modelo en espiral. Informalmente, el riesgo significa sencillamente algo que puede ir mal. Por ejemplo, si la intención es utilizar un nuevo lenguaje de programación, un riesgo es que los compiladores disponibles sean poco fiables o que no produzcan código objeto suficientemente eficiente. Los riesgos originan problemas en el proyecto, como los de confección de agendas y excesos en los costos; por lo tanto, la disminución de riesgos es una actividad muy importante en la gestión del proyecto.
Un ciclo de la espiral empieza con la elaboración de objetivos, como el rendimiento y la funcionalidad. Entonces se enumeran formas alternativas de alcanzar estos objetivos y las restricciones impuestas en cada una de ellas. Cada alternativa se evalúa contra cada objetivo y se identifican las fuentes de riesgo del proyecto. El siguiente paso es resolver estos riesgos mediante actividades de recopilación de información como la de detallar más el análisis, la construcción de prototipos y la simulación. Una vez que se han evaluado los riesgos, se lleva a cabo cierto desarrollo, seguido de una actividad de planificación para la siguiente fase del proceso.

Modelo Espiral WINWIN

El modelo en espiral WINWIN de Boehm, define un conjunto de actividades de negociación al principio de casa paso alrededor de la espiral.

Más que una simple actividad de comunicación con el cliente se definen las siguientes actividades:

- **Identificación** del sistema o subsistemas clave de los directivos.
- **Determinación** de las condiciones de victoria de los directivos.
- **Negociación** de las condiciones de victoria de los directivos para reunirlas en un conjunto de condiciones para todos los afectados (incluyendo el equipo del proyecto de software).

El modelo en espiral WINWIN introduce tres hitos en el proceso, llamados puntos de fijación que ayudan a establecer la completitud de un ciclo alrededor del espiral y proporcionan hitos de decisión antes de continuar el proyecto de software.

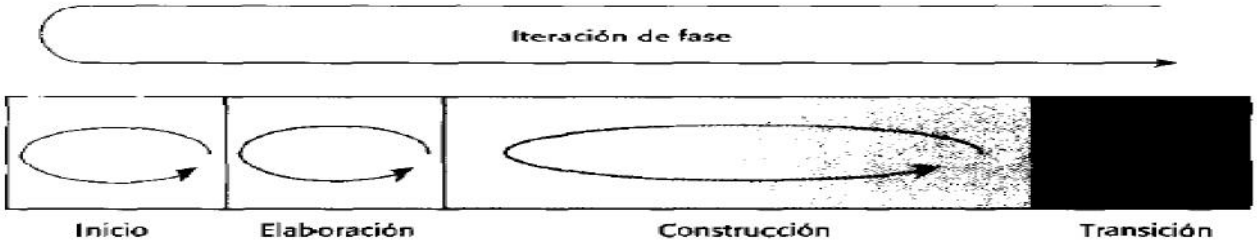
Proceso Unificado de Rational

El Proceso Unificado de Rational (RUP) es un ejemplo de un modelo de proceso moderno que proviene del trabajo en el UML y el asociado Proceso Unificado de Desarrollo de Software.
El RUP reconoce que los modelos de procesos genéricos presentan un sola enfoque del proceso. En contraste, el RUP se describe normalmente desde tres perspectivas:

1. Una perspectiva dinámica que muestra las fases del modelo sobre el tiempo.
2. Una perspectiva estática que muestra las actividades del proceso que se representan.
3. Una perspectiva práctica que sugiere buenas prácticas a utilizar durante el proceso.

La mayor parte de las descripciones del RUP intentan combinar las perspectivas estática y dinámica es un único diagrama. Esto hace el proceso más difícil de entender, por lo que aquí se utilizan descripciones separadas de cada una de estas perspectivas.

El RUP es un modelo en fases que identifica cuatro fases diferentes en el proceso del software.
Sin embargo, a diferencia del modelo en cascada donde las fases se equiparan con las actividades del proceso, las fases en el RUP están mucho más relacionadas con asuntos de negocio más que técnicos. La Figura siguiente se muestra las fases en el RUP.



Estas son:

1. *Inicio*. El objetivo de la fase de inicio es el de establecer un caso de negocio para el sistema. Se deben identificar todas las entidades externas (personas y sistemas) que interactuarán con el sistema y definir estas interacciones. Esta información se utiliza entonces para evaluar la aportación que el sistema hace al negocio. Si esta aportación es de poca importancia, se puede cancelar el proyecto después de esta fase.
2. *Elaboración*. Los objetivos de la fase de elaboración son desarrollar una comprensión del dominio del problema, establecer un marco de trabajo arquitectónico para el sistema, desarrollar el plan del proyecto e identificar los riesgos clave del proyecto. Al terminar esta fase, se debe tener un modelo de los requerimientos del sistema (se especifican los casos de uso UML), una descripción arquitectónica y un plan de desarrollo del software.
3. *Construcción*. La fase de construcción fundamentalmente comprende el diseño del sistema, la programación y las pruebas. Durante esta fase se desarrollan e integran las partes del sistema. Al terminar esta fase, debe tener un sistema software operativo y la documentación correspondiente lista para entregarla a los usuarios.
4. *Transición*. La fase final del RUP se ocupa de mover el sistema desde la comunidad de desarrollo a la comunidad del usuario y hacerlo trabajar en un entorno real. Esto se deja de lado en la mayor parte de los modelos de procesos del software pero es, en realidad, una actividad de alto costo y a veces problemática. Al terminar esta fase, se debe tener un sistema software documentado que funciona correctamente en su entorno operativo.

La iteración dentro del RUP es apoyada de dos formas. Cada fase se puede representar de un modo iterativo con los resultados desarrollados incrementalmente.
Además, el conjunto entero de fases puede también representarse de forma incremental, como se muestra en la citada figura por la flecha en forma de bucle desde la Transición hasta el Inicio.
La vista estática del RUP se centra en las actividades que tienen lugar durante el proceso de desarrollo. Estas se denominan *flujos de trabajo* en la descripción del RUP. Existen seis principales flujos de trabajo del proceso identificados en el proceso y tres principales

flujos de trabajo de soporte. El RUP se ha diseñado conjuntamente con el UML —un lenguaje de modelado, por lo que la descripción del flujo de trabajo se orienta alrededor de los modelos UML asociados.
La ventaja de presentar perspectivas dinámicas y estáticas es que las fases del proceso de desarrollo no están asociadas con flujos de trabajo específicos. Al menos en principio, todos los flujos de trabajo del RUP pueden estar activos en todas las etapas del proceso. Por supuesto, la mayor parte del esfuerzo se realizará en flujos de trabajo tales como el modelado del negocio y los requerimientos en las primeras fases del proceso y en las pruebas y despliegue en las fases posteriores.

La perspectiva práctica en el RUP describe buenas prácticas de la ingeniería del software que son aconsejables en el desarrollo de sistemas. Se recomiendan seis buenas prácticas fundamentales:
1. *Desarrolle el software de forma iterativa*. Planifique incrementos del sistema basado en las prioridades del usuario y desarrollo y entregue las características del sistema de más alta prioridad al inicio del proceso de desarrollo.
2. *Gestione los requerimientos*. Documente explícitamente los requerimientos del cliente y manténgase al tanto de los cambios de estos requerimientos. Analice el impacto de los cambios en el sistema antes de aceptarlos.
3. *Utilice arquitecturas basadas en componentes*. Estructure la arquitectura del sistema en componentes como se indicó anteriormente en este capítulo.
4. *Modele el software visualmente*. Utilice modelos gráficos UML para presentar vistas estáticas y dinámicas del software.
5. *Verifique la calidad del software*. Asegure que el software cumple los estándares de calidad organizacionales.
6. *Controle los cambios del software*. Gestione los cambios del software usando un sistema de gestión de cambios y procedimientos y herramientas de gestión de configuraciones.

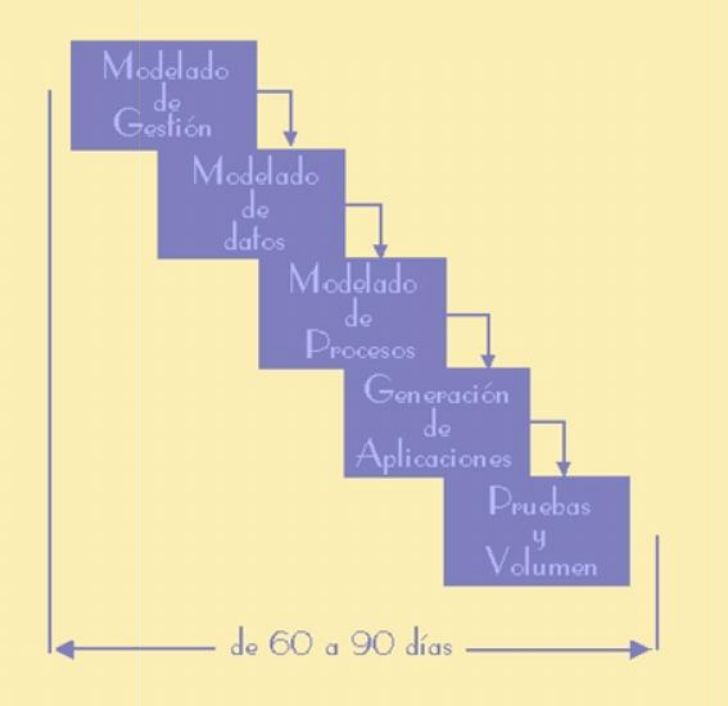
El RUP no es un proceso apropiado para todos los tipos de desarrollo sino que representa una nueva generación de procesos genéricos. Las innovaciones más importantes son la separación de fases y los flujos de trabajo, y el reconocimiento de que la utilización del software en un entorno del usuario es parte del proceso. Las fases son dinámicas y tienen objetivos.
Los flujos de trabajo son estáticos y son actividades técnicas que no están asociadas con fases únicas sino que pueden utilizarse durante el desarrollo para alcanzar los objetivos de cada fase.

Modelo de Desarrollo Rápido de Aplicaciones (Rapid Application Development)

El RAD es un proceso de desarrollo de software, desarrollado inicialmente por James Martin en 1980. El método comprende el desarrollo iterativo, la construcción de prototipos y el uso de utilidades CASE. Tradicionalmente, el desarrollo rápido de aplicaciones tiende a englobar también la usabilidad, utilidad y la rapidez de ejecución.

El Desarrollo Rápido de Aplicaciones (DRA) (Rapid Application Development RAD) es un modelo de proceso del desarrollo del software lineal secuencial que enfatiza un ciclo de desarrollo extremadamente corto. DRA es una adaptación a "Alta velocidad" en el que se logra el desarrollo rápido utilizando un enfoque de construcción basado en componentes. Si se comprenden bien los requisitos y se limita el ámbito del proyecto, el proceso DRA permite al equipo de desarrollo crear un "sistema completamente funcional" dentro de periodos cortos de tiempo. Cuando se utiliza principalmente para aplicaciones de sistemas de información, el enfoque DRA comprende las siguientes fases:

- Modelado de gestión: el flujo de información entre las funciones de gestión se modela de forma que responda a las siguientes preguntas: ¿Qué información conduce el proceso de gestión? ¿Qué información se genera? ¿Quién la genera? ¿A dónde va la información? ¿Quién la proceso?
- Modelado de datos: el flujo de información definido como parte de la fase de modelado de gestión se refina como un conjunto de objetos de datos necesarios para apoyar la empresa. Se definen las características (llamadas atributos) de cada uno de los objetos y las relaciones entre estos objetos.
- Modelado de proceso: los objetos de datos definidos en la fase de modelado de datos quedan transformados para lograr el flujo de información necesario para implementar una función de gestión. Las descripciones del proceso se crean para añadir, modificar, suprimir, o recuperar un objeto de datos. Es la comunicación entre los objetos.
- Generación de aplicaciones: El DRA asume la utilización de técnicas de cuarta generación. En lugar de crear software con lenguajes de programación de tercera generación, el proceso DRA trabaja para volver a utilizar componentes de programas ya existentes (cuando es posible) o a crear componentes reutilizables (cuando sea necesario). En todos los casos se utilizan herramientas automáticas para facilitar la construcción del software.
- Pruebas de entrega: Como el proceso DRA enfatiza la reutilización, ya se han comprobado muchos de los componentes de los programas. Esto reduce tiempo de pruebas. Sin embargo, se deben probar todos los componentes nuevos y se deben ejercitar todas las interfaces a fondo.



Obviamente la limitación de tiempo impuesto en un proyecto DRA demanda "ámbito en escalas". Si una aplicación de gestión puede modularse se forma que permita completarse cada una de las funciones principales en menos de tres meses (utilizando el enfoque descrito anteriormente), es un candidato del DRA. Cada una de las funciones puede ser afrontada por un equipo DRA diferente y ser integradas en un solo conjunto.

Algunos inconvenientes:

- Para proyectos grandes aunque por escalas, el DRA requiere recursos humanos suficientes como para crear el numero correcto de equipos DRA.
- DRA requiere clientes y desarrolladores comprometidos en las rápidas actividades necesarias para completar un sistema en un marco de tiempo abreviado. Si no hay compromiso, por ninguna de las partes constituyentes, los proyectos DRA fracasaran.

No todos los tipos de aplicaciones son apropiados para DRA. Si un sistema no se puede modulizar adecuadamente, la construcción de los componentes necesarios para DRA será problemático. Si está en juego el alto rendimiento, y se va a conseguir el rendimiento convirtiendo interfaces en componentes de sistema, el enfoque DRA puede que no funcione. DRA no es adecuado cuando los riesgos técnicos son altos. Esto ocurre cuando una nueva aplicación hace uso de tecnologías nuevas, o cuando el nuevo software requiere un alto grado de interoperabilidad con programas de computadora ya existentes.

DRA enfatiza el desarrollo de componentes de programas reutilizables. La reutilización es la piedra angular de las tecnologías de objetos, y se encuentra en el modelo de proceso de ensamblaje.

Ventajas de RAD

1. Comprar puede ahorrar dinero en comparación con construir.
2. Los entregables pueden ser fácilmente trasladados a otra plataforma.
3. El desarrollo se realiza a un nivel de abstracción mayor.
4. Visibilidad temprana.
5. Mayor flexibilidad.
6. Menor codificación manual.
7. Mayor involucramiento de los usuarios.
8. Posiblemente menos fallas.
9. Posiblemente menor costo.
10. Ciclos de desarrollo más pequeños.
11. Interfaz gráfica estándar.

Desventajas de RAD

1. Comprar puede ser más caro que construir.
2. Costo de herramientas integradas y equipo necesario.
3. Progreso más difícil de medir.
4. Menos eficiente.
5. Menor precisión científica.
6. Riesgo de revertirse a las prácticas sin control de antaño.
7. Más fallas (por síndrome de “codificar a lo bestia”).
8. Prototipos pueden no escalar, un problema mayúsculo.
9. Funciones reducidas (por “timeboxing”).
10. Dependencia en componentes de terceros: funcionalidad de más o de menos, problemas legales.

Programación Extrema

La programación extrema (XP) es una metodología de desarrollo ágil que tiene como principal objetivo aumentar la productividad a la hora de desarrollar un proyecto software. Da prioridad a los trabajos que dan un resultado directo y en los cuales se reduce la burocracia que pueda existir en el entorno de trabajo.
La efectividad de XP se consigue a través de diversas prácticas de trabajo distintas. El objetivo principal de XP es entregar un software de calidad controlado por las necesidades del cliente. Consigue esté objetivo administrando la complejidad. COmo un sistema que crece en complejidad, el coste para añadir y modificar opciones incrementa. Sin embargo, si el sistema no llega a complicarse tanto, el coste de añadir y modificar opciones más adelante es casi el mismo que el de añadirlas ahora.Ésta es un arma poderosa ya que las metodologías tradicionales suelen seguir la curva de forma que el coste de modificación del software incrementa exponencialmente a medida que se invierte más tiempo en todas las fases del desarrollo.

Valores de la programación extrema

Para garantizar el éxito de un proyecto, los autores de XP han considerado como fundamentales cuatro valores:

- **Comunicación**. Muy importante. La XP ayuda mediante sus prácticas a la comunicación entre los integrantes del grupo de trabajo: jefes de proyecto, clientes y desarrolladores.
- **Sencillez**. Los programas deben ser los más sencillos posibles y tener la funcionalidad necesaria que se indican en los requisitos. No hay que añadir algo que no se necesite *hoy*. Si se necesita añadir más funcionalidad *mañana* pues ya se hará entonces.
- **Retroalimentación**. Las pruebas que se le realizan al software nos mantiene informados del grado de fiabilidad del sistema.
- **Valentía**. Asumir retos, ser valientes ante los problemas y afrontarlos. El intentar mejorar algo que ya funciona. Aunque gracias a las pruebas unitarias no existe el riesgo de 'meter la pata'.

Algunas voces, añaden además un quinto valor: la humildad. Con la compartición de código, la refactorización y el trabajo de equipo tan estrecho una buena dosis de humildad siempre es de agradecer.

Prácticas de la programación extrema

- **El juego de la planificación** (the planning game). Es un permanente diálogo entre las partes empresarial (deseable) y técnica (posible).
- **Pequeñas entregas** (small releases). Cada versión debe de ser tan pequeña como fuera posible, conteniendo los requisitos de negocios más importantes, las versiones tiene que tener sentido como un todo, me explico no puedes implementar media característica y lanzar la versión.
  Es mucho mejor planificar para 1 mes o 2 que para seis meses y un año, las compañías que entregan software muy voluminoso no son capaces de hacerlo con mucha frecuencia.
- **Metáfora** (metaphor). Una metáfora es una historia que todo el mundo puede contar a cerca de cómo funciona el sistema. Algunas veces podremos encontrar metáforas sencillas "Programa de gestión de compras, ventas, con gestión de cartera y almacén". Las metáforas ayudan a cualquier persona a entender el objeto del programa.
- **Diseño sencillo** (simple design). Cuando implementamos nuevas características en nuestros programas nos planteamos la manera de hacerlo lo mas simple posible, después de implementar esta característica, nos preguntamos como hacer el programa mas simple sin perder funcionalidad, este proceso se le denomina recodificar o refactorizar (refactoring). Esto a veces nos puede llevar a hacer mas trabajo del necesario, pero a la vez estaremos preparando nuestro sistema para que en un futuro acepte nuevos cambios y pueda albergar nuevas características. No debemos de recodificar ante especulaciones si no solo cuando el sistema te lo pida.

- **Pruebas** (testing). No debe existir ninguna característica en el programa que no haya sido probada, los programadores escriben pruebas para chequear el correcto funcionamiento del programa, los clientes realizan pruebas funcionales. El resultado un programa mas seguro que conforme pasa el tiempo es capaz de aceptar nuevos cambios.
- **Refactorización** (refactoring). Cuando implementamos nuevas características en nuestros programas nos planteamos la manera de hacerlo lo mas simple posible, después de implementar esta característica, nos preguntamos como hacer el programa mas simple sin perder funcionalidad, este proceso se le denomina recodificar o refactorizar (refactoring). Esto a veces nos puede llevar a hacer mas trabajo del necesario, pero a la vez estaremos preparando nuestro sistema para que en un futuro acepte nuevos cambios y pueda albergar nuevas características. No debemos de recodificar ante especulaciones si no solo cuando el sistema te lo pida.
- **Programación por parejas** (pair programming). Todo el código de producción lo escriben dos personas frente al ordenador, con un sólo ratón y un sólo teclado. Cada miembro de la pareja juega su papel: uno codifica en el ordenador y piensa la mejor manera de hacerlo, el otro piensa mas estratégicamente, ¿Va a funcionar?, ¿Puede haber pruebas donde no funcione?, ¿Hay forma de simplificar el sistema global para que el problema desaparezca?
  El emparejamiento es dinámico, puedo estar emparejado por la mañana con una persona y por la tarde con otra, si tienes un trabajo sobre un área que no conoces muy bien puedes emparejarte con otra persona que si conozca ese área. Cualquier miembro del equipo se puede emparejar con cualquiera.
- **Propiedad colectiva** (collective ownership). Cualquiera que crea que puede aportar valor al código en cualquier parcela puede hacerlo, ningún miembro del equipo es propietario del código. Si alguien quiere hacer cambios en el código puede hacerlo. Si hacemos el código propietario, y necesitamos de su autor para que lo cambie entonces estaremos alejándonos cada vez mas de la comprensión del problema, si necesitamos un cambio sobre una parte del código lo hacemos y punto. XP propone un propiedad colectiva sobre el código nadie conoce cada parte igual de bien pero todos conoce algo sobre cada parte, esto nos preparará para la sustitución no traumática de cada miembro del equipo.
- **Integración continua** (continuos integration). El código se debe integrar como mínimo una vez al día, y realizar las pruebas sobre la totalidad del sistema. Una pareja de programadores se encargara de integrar todo el código en una maquina y realizar todas las pruebas hasta que estas funcionen al 100%.
- **40 horas semanales** (40-hour week). Si queremos estar frescos y motivados cada mañana y cansado y satisfecho cada noche. El viernes quiero estar cansado y satisfecho para sentir que tengo dos días para pensar en algo distinto y volver el lunes lleno de pasión e ideas. Esto requiere que trabajemos 40 horas a la semana, mucha gente no puede estar más de 35 horas concentrada a la semana, otros pueden llegar hasta 45 pero ninguno puede llegar a 60 horas durante varias semanas y aun seguir fresco, creativo y confiado. Las horas extras son síntoma de serios problemas en el proyecto, la regla de XP dice nunca 2 semanas seguidas realizando horas extras.

- **Cliente en casa** (on-site costumer). Un cliente real debe sentarse con el equipo de programadores, estar disponible para responder a sus preguntas, resolver discusiones y fijar las prioridades. Lo difícil es que el cliente nos ceda una persona que conozca el negocio para que se integre en el equipo normalmente estos elementos son muy valiosos, pero debemos de hacerles ver que será mejor para su negocio tener un software pronto en funcionamiento, y esto no implica que el cliente no pueda realizar cualquier otro trabajo.
- **Estándares de codificación** (coding standards). Si los programadores van a estar tocando partes distintas del sistema, intercambiando compañeros, haciendo refactoring, debemos de establecer un estándar de codificación aceptado e implantado por todo el equipo.

Algunas de estas prácticas reciben muchas críticas, sobre todo la programación por parejas. Muchos jefes de proyecto no aceptan que dos desarrolladores tengan un único ordenador, ya que creen que esto minimizará la productividad, sin saber que de este modo puede producirse el mismo código que dos personas trabajando en solitario pero con un código de mayor calidad.
Se dice que si no se realizan estas doce prácticas no se está realizando XP. Yo, a mi modo de verlo, creo que nosotros no tenemos que adaptarnos a ninguna metodología, sino que ellas se tienen que adaptar a nosotros dependiendo del tipo de proyecto que estemos desarrollando.

Herramientas de la XP

Historias de usuarios

Son tarjetas físicas en las cuales se anota una descripción de una funcionalidad del sistema, en una oración, se le da un número y un título para ser identificada.
Casos de prueba de aceptación

Son tarjetas que se elaboran para realizar las pruebas de cada historia de usuario.

Tarea de ingeniería

Son tarjetas que se elaboran para ayudar y simplificar la programación de una historia de usuario.

Tarjetas CRC

Describen las clases utilizadas en la programación de una historia.
Ventajas y desventajas
Una de las ventajas de la programación extrema es que se adapta al desarrollo de sistemas pequeños y grandes; optimiza el tiempo de desarrollo; permite realizar el desarrollo del sistema en parejas para complementar los conocimientos; el código es sencillo y entendible, además de la poca documentación a elaborar para el desarrollo del sistema.
Las desventajas son que no se tiene la definición del costo y el tiempo de desarrollo; el

sistema va creciendo después de cada entrega al cliente y nadie puede decir que el cliente no querrá una función más; se necesita de la presencia constante del usuario, lo cual en la realidad es muy difícil de lograr.
Cuándo utilizar XP
Es mejor utilizar XP cuándo el proyecto en cuestión tiene un alto riesgo de elementos añadidos. Quizá sea satisfacer un plazo muy ajustado. Quizá sean alguna cantidad o medio de requerimiento dinámico desconocido que la solución propuesta no tiene garantías de satisfacer. Cabe la posibilidad de que la solución propuesta sea tan avanzada que corre el riesgo simplemente de no funcionar.
El equipo de desarrollo real debe ser pequeño (no menos de dos personas y no más de 10 u 11) y todos los miembros del equipo de proyecto (no sólo los de desarrollo) pueden apropiarse del proceso de creación.

Ventajas:

- Se incrementa considerablemente la productividad del equipo de desarrollo.
- Se consigue con más facilidad la satisfacción del cliente.
- Se satisfacen y mantienen los estándares de calidad.
- Se satisfacen con más precisión los requisitos del cliente.
- Se cumplen los plazos con mayor exactitud y consistencia.

Existe mucho debate entre la comunidad de programación sobre el éxito relativo y la fiabilidad de XP. Con frecuencia las enormes presiones comerciales significan que simplemente es un paradigma imposible de seguir. Sin embargo, cómo en todo paradigma, es simplemente un modelo (algo a lo que se pretende llegar). Si te gustaría investigar más sobre esté interesante tema y encontrar recomendaciones y buenas prácticas sobre XP visita su sitio Web.

Introducción al KanBan

En la actualidad, si una empresa no es lo suficientemente flexible para adaptarse a los cambios del mercado se podría decir que esa empresa estará fuera de competencia en muy poco tiempo.
¿Que es ser flexible?, de acuerdo a su definición literal es "Que se puede doblar fácilmente, que se acomoda a la dirección de otro", esto aplicado a la manufactura se traduciría, "que se acomoda a las necesidades y demanda del cliente", tanto de diseño, calidad y entrega.
Uno de las problemáticas más comunes en lo que respecta a la planeación de la producción es producir lo necesario en el tiempo necesario, sin sobrantes ni faltantes, para lograr esto se necesita un plan, un plan flexible, un plan hecho para ser modificado, un plan que se pueda modificar rápidamente.
Un plan de producción es influenciado tanto externamente como internamente. Las condiciones del mercado cambian constantemente. Para responder a estos cambios, se deben dar instrucciones constantemente al área de trabajo.
Ya que queremos producir en un sistema Justo a Tiempo, las instrucciones de trabajo deben ser dadas de manera constante en intervalos de tiempo variados.
La información más importante en el área de trabajo será, cuanto debemos producir de

cual producto en ese momento; las instrucciones pueden ser dadas como se van necesitando.

Ya que no es conveniente hacer órdenes de producción muy grandes tratando de prevenir la demanda del mercado ya que nos podemos quedar cortos o largos de producto, así como no es conveniente hacer órdenes unitarias; lo más conveniente es hacer órdenes de lotes pequeños, este es el concepto fundamental.

Es muy importante que los trabajadores sepan qué están produciendo, qué características lleva, así como qué van a producir después y que características tendrá.

Muchas compañías manufactureras japonesas visualizaron el ensamble de un producto como continua desde el Diseño-Manufactura-Distribución de Ventas-Servicio al Cliente. Para muchas compañías del Japón el corazón de este proceso antes mencionado es el Kanban, quien directa o indirectamente maneja mucho de la organización manufacturera. Fue originalmente desarrollado por Toyota en la década de los 50 como una manera de manejo del flujo de materiales en una línea de ensamble.

Sobre las pasadas tres décadas el proceso Kanban que se define como "Un sistema de producción altamente efectivo y eficiente" ha desarrollado un ambiente de óptimo manufacturero envuelto en competitividad global.

Es muy importante que los trabajadores sepan qué están produciendo, qué características lleva, así como qué van a producir después y que características tendrá.

Para muchas compañías del Japón el corazón de este proceso es el Kanban, quien directa o indirectamente maneja mucho de la organización manufacturera.

Nuestros objetivos específicos son los siguientes:

- En una empresa manufacturara, poder empezar cualquier operación estándar en cualquier momento.
- Dar instrucciones de la producción basados en las condiciones actuales del área de trabajo.
- Prevenir que en las organizaciones se agregue trabajo innecesario a aquellas órdenes ya empezadas, y evitar el exceso de papeleo innecesario.
- Propender a la eliminación de la sobreproducción.
- Facilitar el control del material.

Metodología: Desarrollo de Software Adaptable

El desarrollo de software adaptable (ASD) es una metodología de desarrollo impulsada por Jim Highsmith y San Bayer que hace énfasis en aplicar las ideas que se originaron en el mundo de los sistemas complejos, adaptación continua del proceso al trabajo.

El ASD es iterativo orientado a los componentes de software (la funcionalidad que el producto va a tener, características, etc.) mas que que a las tareas en las que se va a alcanzar dicho objetivo.

Es tolerante a cambios. La revisión de los componentes sirve para aprender de los errores y volver a iniciar el ciclo de desarrollo.

METODOS FORMALES

La denominación métodos formales se usa para referirse a cualquier actividad relacionada con representaciones matemáticas del software, incluyendo la especificación formal de sistemas, análisis y demostración de la especificación, el desarrollo

transformacional y la verificación de programas. Todas estas actividades dependen de una especificación formal del software.

Una especificación formal del software es una especificación expresada en un lenguaje cuyo vocabulario, sintaxis y semántica están formalmente definidos. Esta necesidad de unadefinición formal significa que los lenguajes de especificación deben basarse en conceptos matemáticos cuyas propiedades se comprendan bien. La rama de las matemáticas usada es la de matemática discreta, y los conceptos matemáticos provienen de la teoría de conjuntos, la lógica y el álgebra.

En la década de los 80, muchos investigadores de ingeniería del software propusieron que el uso de métodos formales de desarrollo era la mejor forma de mejorar la calidad del software. Argumentaban que el rigor y el análisis detallado, que son una parte esencial de los métodos formales, podrían dar lugar a programas con menos errores y más adecuados a las necesidades de los usuarios. Predijeron que, en el siglo XXI, una gran proporción del software estaría desarrollado usando métodos formales.

Claramente, esta predicción no se ha hecho realidad. Existen cuatro razones principales para esto:

1. Una ingeniería del software exitosa. El uso de otros métodos de ingeniería del software como los métodos estructurados, gestión de configuraciones y ocultación de la información en el diseño del software y procesos de desarrollo ha conducido a mejoras en la calidad del software. La gente que sugirió que la única forma de mejorar la calidad del software era usando métodos formales estaba claramente equivocada.

2. Cambios en el mercado. En la década de los 80, la calidad del software fue vista como un problema clave de la ingeniería del software. Sin embargo, desde entonces, la cuestión crítica para muchas clases de desarrollo del software no es la calidad, sino la oportunidad de mercado. El software debe desarrollarse rápidamente, y los clientes están dispuestos a aceptar software con algunos defectos si se les entrega rápidamente. Las técnicas para el desarrollo rápido del software no funcionan de forma efectiva con las especificaciones formales. Por supuesto, la calidad todavía es un factor importante, pero debe lograrse en el contexto de entrega rápida.

3. Ámbito limitado de los métodos formales. Los métodos formales no son muy apropiados para la especificación de interfaces de usuario e interacciones del usuario. El componente de interfaz de usuario se ha convertido en una parte cada vez mayor de la mayoría de los sistemas, de manera que realmente s6lo pueden usarse métodos formales cuando se desarrollan las otras partes del sistema.

4. Escalabilidad limitada de los métodos formales. Los métodos formales todavía no son muy escalables. La mayoría de los proyectos con éxito que han usado estas técnicas han estado relacionados con núcleos de sistemas críticos relativamente pequeños. A medida que los sistemas incrementan su tamaño, el tiempo y esfuerzo requerido para desarrollar una especificación formal crece de forma desproporcionada.

Ventajas

- Se comprende mejor el sistema.
- La comunicación con el cliente mejora ya que se dispone de una descripción clara y no ambigua de los requisitos del usuario.

- El sistema se describe de manera más precisa.
- El sistema se asegura matemáticamente que es correcto según las especificaciones.
- Mayor calidad software respecto al cumplimiento de las especificaciones.
- Mayor productividad

Desventajas

- El desarrollo de herramientas que apoyen la aplicación de métodos formales es complicado y los programas resultantes son incómodos para los usuarios.
- Los investigadores por lo general no conocen la realidad industrial.
- Es escasa la colaboración entre la industria y el mundo académico, que en ocasiones se muestra demasiado dogmático.
- Se considera que la aplicación de métodos formales encarece los productos y ralentiza su desarrollo.

El uso de métodos formales está creciendo en el área del desarrollo de sistemas críticos, en donde las propiedades emergentes del sistema tales como seguridad, fiabilidad y protección son muy importantes. El alto coste de los fallos de funcionamiento en estos sistemas implica que las compañías están dispuestas a aceptar los costes elevados iniciales de los métodos formales para asegurar que su software es tan confiable como sea posible.
Los sistemas críticos en los que los métodos formales se han aplicado con éxito incluyen un sistema de información de control de tráfico aéreo, sistemas de señalización de redes ferroviarias, sistemas de naves espaciales y sistemas de control médico.
Clasificación
La clasificación más común se realiza en base al modelo matemático subyacente en cada método, de esta manera podrían clasificarse en:

- *Especificaciones basadas en lógica de primer orden y teoría de conjuntos*: permiten especificar el sistema mediante un concepto formal de estados y operaciones sobre estados. Los datos y relaciones/funciones se describen en detalle y sus propiedades se expresan en lógica de primer orden. La semántica de los lenguajes está basada en la teoría de conjuntos. Los métodos de este tipo más conocidos son: Z, VDM y B.
- *Especificaciones algebraicas*: proponen una descripción de estructuras de datos estableciendo tipos y operaciones sobre esos tipos. Para cada tipo se define un conjunto de valores y operaciones sobre dichos valores. Las operaciones de un tipo se definen a través de un conjunto de axiomas o ecuaciones que especifican las restricciones que deben satisfacer las operaciones. Métodos más conocidos: Larch, OBJ, TADs.
- *Especificación de comportamiento:*
- *Métodos basados en álgebra de procesos*: modelan la interacción entre procesos concurrentes. Esto ha potenciado su difusión en la especificación de sistemas de comunicación (protocolos y servicios de telecomunicaciones) y de sistemas distribuidos y concurrentes. Los más conocidos son: CCS,CSP y LOTOS.
- *Métodos basados en Redes de Petri*: una red de petri es un formalismo basado en autómatas, es decir, un modelo formal basado en flujos de información. Permiten expresar eventos concurrentes. Los formalismos basados en redes de petri establecen la

noción de estado de un sistema mediante lugares que pueden contener marcas. Un conjunto de transiciones (con pre y post condiciones) describe la evolución del sistema entendida como la producción y consumo de marcas en varios puntos de la red.

- *Métodos basados en lógica temporal*: se usan para especificar sistemas concurrentes y reactivos. Los sistemas reactivos son aquellos que mantienen una continua interacción con su entorno respondiendo a los estímulos externos y produciendo salidas en respuestas a los mismos, por lo tanto el orden de los eventos en el sistema no es predecible y su ejecución no tiene por qué terminar.

Ingeniería del Software Asistida por computadora

Ingeniería del Software Asistida por Computadora (CASE) es el nombre que se le da al software que se utiliza para ayudar a las actividades del proceso del software como la ingeniería de requerimientos, el diseño, el desarrollo de programas y las pruebas. Por lo tanto, las herramientas CASE incluyen editores de diseño, diccionarios de datos, compiladores, depuradores, herramientas de construcción de sistemas, etcétera.
La tecnología CASE proporciona ayuda al proceso del software automatizando algunas de sus actividades, así como proporcionando información acerca del software en desarrollo. Algunos ejemplos de las actividades que se pueden automatizar utilizando CASE son:

1. El desarrollo de modelos gráficos del sistema como parte de la especificación de requerimientos o del diseño de software.
2. La comprensión del diseño utilizando un diccionario de datos que tiene información sobre las entidades y relaciones del diseño.
3. La generación de interfaces de usuario a partir de la descripción gráfica de la interfaz que es elaborada de forma interactiva por el usuario.
4. La depuración de programas por medio de la provisión de la información proporcionada por los programas en ejecución.
5. La conversión automática de programas de una versión anterior de una lenguaje de programación, a una versión más reciente.

La tecnología CASE está disponible para la mayoría de las actividades rutinarias en el proceso del software. Esto permite algunas mejoras en la calidad y productividad del software, aunque éstas sean menores que las predichas por los primeros partidarios de CASE. Estos sugirieron que se tendría una mejora mayor si se utilizaran entornos CASE integrados.
Aunque esto es significante, las predicciones que se hicieron cuando se introdujeron las herramientas CASE en los años 80 y 90 fueron que el uso de la tecnología CASE generaría enormes ahorros en los costos del proceso del software.

Las mejoras por la utilización de CASE están limitadas por dos factores:

1. Esencialmente, la ingeniería del software es una actividad de diseño que se basa en la creatividad. Los sistemas CASE automatizan las actividades rutinarias, pero los intentos de utilizar la inteligencia artificial para proporcionar ayuda al diseño no han tenido éxito.
2. En la mayoría de las organizaciones, la ingeniería del software es una actividad de equipo, y los ingenieros invierten mucho tiempo interactuando con los otros miembros del equipo, La tecnología CASE no proporciona mucha ayuda para esto.

Actualmente, la tecnología CASE está madura y hay herramientas disponibles y bancos de trabajo de un amplio rango de proveedores.

Clasificación de CASE

Las clasificaciones de CASE nos ayudan a comprender los tipos de herramientas CASE y su papel en la ayuda a las actividades de proceso del software. Existen varias formas diferentes de clasificar las herramientas CASE, cada una de las cuales nos proporciona una perspectiva distinta de estas herramientas.

Dichas herramientas desde tres de estas perspectivas:

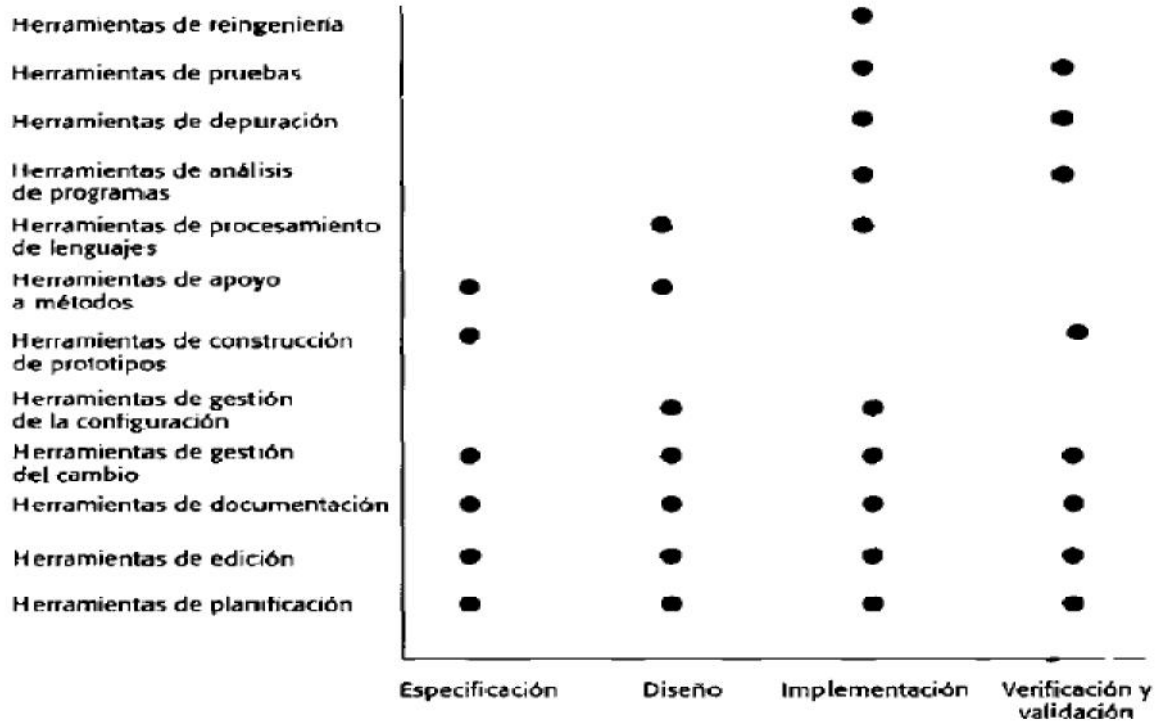
1. *Una perspectiva funcional* en la que las herramientas CASE se clasifican de acuerdo con su función específica.
2. *Una perspectiva de proceso* en la que las herramientas se clasifican de acuerdo con las actividades del proceso que ayudan.
3. *Una perspectiva de integración* en la que las herramientas CASE se clasifican de acuerdo con la forma en que están organizadas en unidades integradas que proporcionan ayuda a una o más actividades del proceso.

La figura siguiente muestra es una clasificación de las herramientas CASE acorde con su función.

Herramientas de planificación	Herramientas PERT, herramientas de estimación, hojas de cálculo.
Herramientas de edición	Editores de texto, editores de diagramas, procesadores de texto.
Herramientas de gestión del cambio	Herramientas de rastreo de requerimientos, sistemas de control de cambios.
Herramientas de gestión de la configuración	Sistema de gestión de las versiones, herramientas de construcción de sistemas.
Herramientas de construcción de prototipos	Lenguajes de muy alto nivel, generadores de interfaz de usuario.
Herramientas de apoyo a métodos	Editores de diseño, diccionarios de datos, generadores de código.
Herramientas de procesamiento de lenguajes	Compiladores, intérpretes.
Herramientas de análisis de programas	Generadores de referencias cruzadas, analizadores estáticos, analizadores dinámicos.
Herramientas de pruebas	Generadores de pruebas de datos, comparadores de archivos.
Herramientas de depuración	Sistemas de depuración interactiva.
Herramientas de documentación	Programas de diseño de páginas, editores de imágenes.
Herramientas de reingeniería	Sistemas de referencias cruzadas, sistemas reestructuración de programas.

Esta tabla enumera diferentes tipos de herramientas CASE y da ejemplos específicos de cada una.

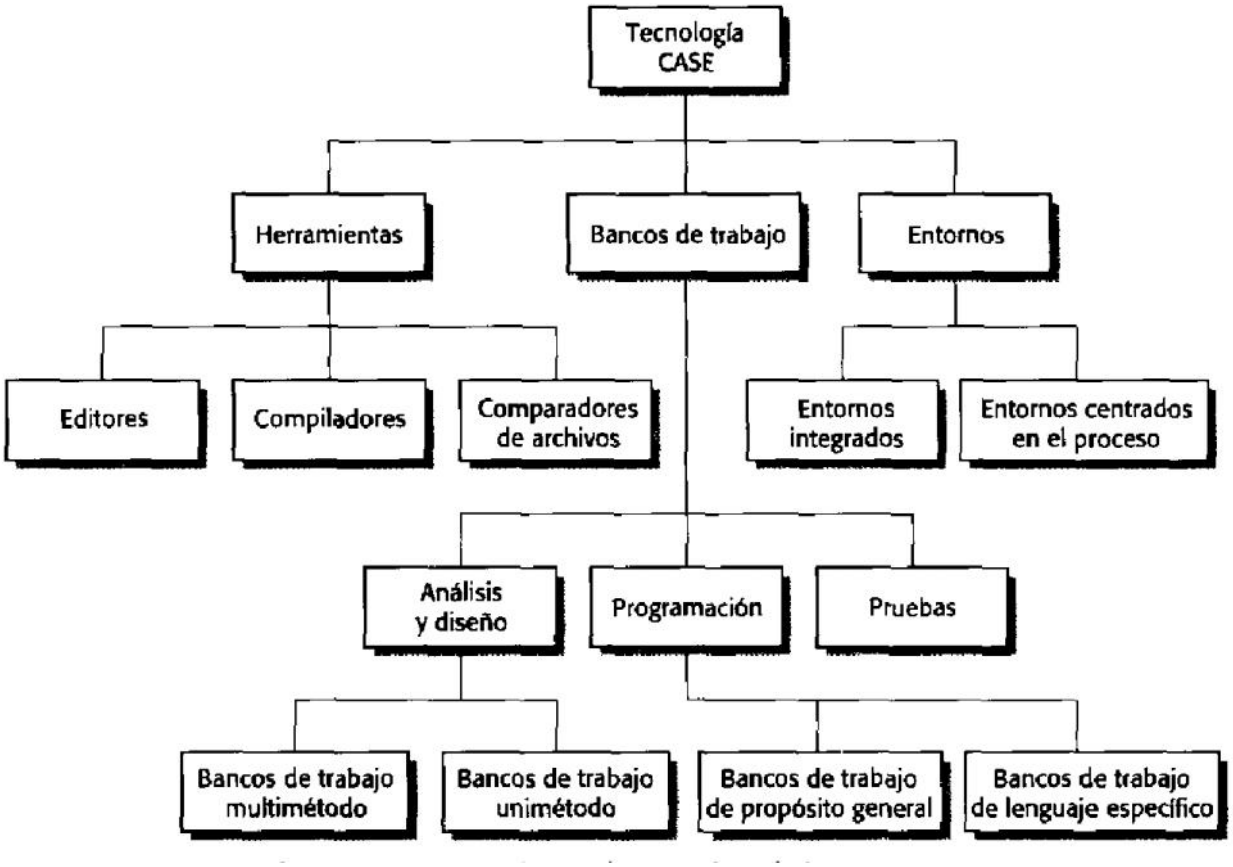
La figura siguiente presenta una clasificación alternativa de las herramientas CASE. Muestra las fases del proceso que reciben ayuda por varios tipos de herramientas CASE.



Las herramientas para la planificación y estimación, edición de texto, preparación de documentos y gestión de la configuración pueden utilizarse durante todo el proceso del software.
Otra dimensión de clasificación posible es la amplia ayuda que ofrece la tecnología CASE para el proceso del software propone que los sistemas CASE se deben clasificar en tres categorías:

1. *Las herramientas ayudan* a las tareas individuales del proceso como la verificación de la consistencia de un diseño, la compilación de un programa y la comparación de los resultados de las pruebas. Las herramientas pueden ser de propósito general, independientes (por ejemplo, un procesador de texto) o agrupadas en bancos de trabajo.
2. *Los bancos de trabajo ayudan* a las fases o actividades del proceso como la especificación, el diseño, etcétera. Normalmente consisten en un conjunto de herramientas con algún grado mayor o menor de integración.
3. *Los entornos ayudan* a todos los procesos del software, o al menos a una parte sustancial de éstos. Normalmente incluyen varios bancos de trabajo integrados.

La figura siguiente ilustra esta clasificación y muestra algunos ejemplos de estas clases de ayuda CASE.



Por supuesto, esto es un ejemplo ilustrativo; muchos tipos de herramientas y bancos de trabajo se han quedado fuera de este diagrama.
Las herramientas de propósito general se utilizan a discreción del ingeniero de software quien toma decisiones acerca de cuándo aplicarlas para ayudar al proceso. Sin embargo, los bancos de trabajo por lo general ayudan a algún método que incluye un modelo del proceso y un conjunto de reglas/pautas que se aplican al software en desarrollo. Los entornos se clasifican en integrados y centrados en el proceso. Los entornos integrados proporcionan ayuda a los datos, al control y a la integración de la presentación. Los entornos centrados en procesos son más generales. Incluyen el conocimiento del proceso del software y un motor de procesos que utiliza este modelo del proceso para aconsejar a los ingenieros sobre qué herramientas o bancos de trabajo hay que aplicar y cuándo deben utilizarse.
En la práctica, los límites entre estas diferentes clases son borrosos. Las herramientas se pueden vender como productos individuales, pero pueden proporcionar ayuda a diferentes actividades.
La clasificación proporciona un primer paso útil para ayudar a entender la amplitud del soporte que una herramienta proporciona al proceso.

Tecnologías emergentes de la información y desarrollo de software

En la actualidad se viven constantes cambios en el ámbito de las Tecnologías de la Información; una de las ramas que más cambios tiene es la Ingeniería de Software. Como describe Román (2011) la Web ha evolucionado de manera significativa pasando por 3 etapas hasta ahora. La primera fue la Web 1.0 las cuales estaban accesibles a personas especializadas que eran las encargadas de crear y mantener el contenido para ser consultado en la Web. Cuando se desarrollaron plataformas para que cualquier persona creara contenido Web sin conocimientos especializados o técnicos y se crearon las primeras redes sociales se hablaba de la evolución de la Web 1.0 a la denominada Web 2.0. Actualmente se habla de una nueva evolución llamada Web 3.0 o Web Semántica que trata de dotar de significado a los recursos existentes en la Web.

Para llevar a cabo los desarrollos tecnológicos que se requieren en esta evolución e innovación constante la Ingeniería en Tecnologías de la Información y Desarrollo de Software se apoyan de otras áreas derivadas como son los Sistemas Embebidos, Cómputo Móvil y Ubicuo, Interfaces Humano-Computadora, Sistemas Distribuidos y Bases de Datos No Relacionales, Redes de Próxima Generación, Arquitectura de Software, Patrones de diseño, Calidad en el Desarrollo de Software y Desarrollo Dirigido por Pruebas, Desarrollo de Software Orientado a Servicios, Modelos y Metodologías de Desarrollo de Software como base para converger y crear las aplicaciones y dispositivos electrónicos necesarios para responder a la constante demanda de información y conocimiento que las personas requieren.

Algunas de las tecnologías más relevantes que se enfocan en el área de Tecnología de la Información y Desarrollo de Software son las siguientes.

Sistemas Embebidos

Según Calva et. al. (2012) los sistemas embebidos son aquellos sistemas informáticos que forman parte de sistemas más grandes y que llevan a cabo algunos de los requisitos necesarios para la operatividad de estos sistemas. Los sistemas integrados cubren una amplia gama de sistemas informáticos de pequeños dispositivos basados en computadoras de ultra a grandes sistemas de vigilancia y control de procesos complejos.

La inmensa mayoría de los sistemas informáticos pertenece a los sistemas integrados. La mayoría de estos sistemas integrados se caracterizan también como sistemas de tiempo real, lo que significa que las propiedades de tiempo real, tales como el tiempo de respuesta, tiempo de ejecución son importantes problemas de diseño. El aumento de la complejidad de los sistemas embebidos de tiempo real conduce a la creciente demanda en relación con la ingeniería de requisitos, diseño de alto nivel, la detección temprana de errores, la productividad, la integración, verificación y mantenimiento, lo que aumenta la importancia de una gestión eficiente de las propiedades del ciclo de vida tales como la mantenibilidad, portabilidad y capacidad de adaptación así como la calidad implementada en los procesos de desarrollo de software.

En la actualidad se han considerado un sin número de procesos para ser automatizados, en su mayoría mediante el control de un software o en su defecto por dispositivos que integren software embebido para su manipulación. Es por ello que cada vez más la

Ingeniería de software está trabajando en mejorar la calidad de los productos y procesos para hacer que dichos sistemas de control y automatización no contengan errores.

Cómputo Móvil y Ubicuo

Hoy en día tiene una relevancia cada vez mayor el uso de dispositivos electrónicos inmersos en nuestras vidas. Esto se debe en parte a los diferentes dispositivos que se interconectan en nuestro entorno ya sea en la oficina, escuela, hogar, o en el transporte público.

La tecnología ubícua permite a los individuos aprender allí donde estén, y contar para ello con los componentes de su entorno social. En entornos de aprendizaje en línea es importante acceder a los recursos de manera tal que los individuos no tengan que preocuparse por la forma o dispositivos que se requieren para conectarse y consumir los objetos de aprendizaje de las plataformas en línea.

Una de las principales características del cómputo móvil y ubicuo es acceder a los recursos a los que se tienen acceso mediante los dispositivos conectados a internet sin siquiera pensar de qué manera estamos logrando la comunicación entre los equipos o dispositivos.

Esta área de conocimiento de las Tecnologías de la Información está cobrando cada vez más importancia ya que continuamente se están mejorando las plataformas y tecnologías de interconexión o redes de próxima generación.

La computación móvil y ubicua está usándose junto a los sistemas distribuidos, sistemas embebidos entre otros para hacer realidad un nuevo concepto tecnológico denominado: Internet de las Cosas (Internet of Things) la cual define como una red de datos que permite la consulta de la información acerca de los objetos del mundo real por medio de un identificador único llamado Código Electrónico de Producto y un mecanismo de resolución. El internet de las cosas utiliza técnicas de la web semántica para dotar de significado a los objetos y datos obtenidos de ellos y al hablar de objetos se entiende que son dispositivos electrónicos.

Redes de próxima Generación

La sociedad en la que actualmente vivimos demanda cada vez más conectividad de los dispositivos electrónicos para acceder a los medios de información, recursos disponibles en internet así como la interconectividad entre sus pares. Es por ello que ahora se habla de la sociedad de la información y del conocimiento, es el proceso que se realimenta a si mismo donde las nuevas tecnologías facultan a la sociedad en el manejo de grandes volúmenes de información, las cuales a su vez, generan más conocimiento en un círculo virtuoso ascendente de progreso.

La organización UIT-T Y (2014) define a estas redes en una red basada en paquetes que permite prestar servicios de telecomunicación y en la que se pueden utilizar múltiples tecnologías de transporte de banda ancha propiciadas por la QoS (Quality of Service), y

en la que las funciones relacionadas con los servicios son independientes de las tecnologías subyacentes relacionadas con el transporte. Permite a los usuarios el acceso sin trabas a redes y a proveedores de servicios y/o servicios de su elección.

La diferencia de las redes especializadas en proveer un servicio específico como las actuales, las NGN son una red multiservicio. Las redes de la próxima generación (NGN) anuncian el paso al enfoque de muchos servicios a través de una sola red, es básicamente una red que unifica voz, datos y video bajo la plataforma IP convirtiéndola en una red en la cual es posible prestar diferentes servicio.

Interfaces Humano Computadora

La interacción Humano-Computadora se encarga de estudiar la relación de la interacción entre el Hombre y las computadoras, buscando hacerla entendible y fácil de usar. Con esta definición se infiere que cualquier dispositivo electrónico que interactúe con un humano ya sea mediante Hardware o Software cubre la misma.

El objetivo primordial de la interacción de usuarios de dispositivos electrónicos es ser eficiente, para ello dberá minimizar los errores, incrementar la satisfacción del usuario, disminuir la frustración referente al uso del dispositivo. En resumen debe hacer más eficientes y productivas las tareas rutinarias y de trabajo de las personas con los dispositivos electrónicos.

Para el desarrollo de nuevas tecnologías o plataformas, lenguajes de programación o software es importante tomar en cuenta la Interfaces que estarán en contacto con los usuarios finales ya que de ello depende en gran medida el éxito o fracaso del sistema de software o hardware que interactúa con el usuario final. Cada vez más las empresas de software se están fijando en dos conceptos que se relacionan entre si los cuales son la ***usabilidad*** y ***accesibilidad***.

La usabilidad se refiere al alcance en el que un producto puede ser utilizado por usuarios específicos para alcanzar metas específicas y la accesibilidad es la posibilidad de que un producto o servicio pueda ser accedido y usado de forma independiente de las limitaciones propias del individuo o de las derivadas del contexto de us. La usabilidad en una aplicación cumple con su función principal siempre que se utilice con efectividad, eficiencia y satisfacción en un contexto específico de uso y la accesibilidad debe de hacer posible usar esa aplicación sin importar las capacidades diferentes del individuo ya sea visual, motriz o alguna otra.

Sistemas Distribuidos y Bases de Datos No Relacionales

Las aplicaciones tienen que almacenar una vasta cantidad de información, esto se debe a la evolución constante de la Web. Al comenzar las redes sociales y la Web 2.0 se comenzaron a diseñar aplicaciones basadas en sistemas Distribuidos ya que la velocidad de respuesta debe ser mayor a la que ofrece un servidor central de aplicaciones. Los sistemas distribuidos ahora son muy utilizados sobre todo por el nuevo concepto BigData el cual es la tendencia en el avance de la tecnología que ha abierto las puertas hacia un

nuevo enfoque de entendimiento y toma de decisiones, la cual es utilizada para describir enormes cantidades de datos (estructurados, no estructurados y semiestructurados) que tomaría demasiado tiempo y sería muy costoso cargarlos a un base de datos relacional para su análisis.

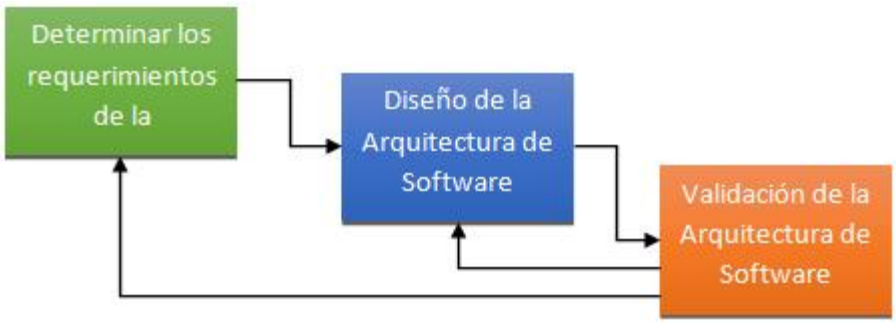
De tal manera el concepto de Big Data aplica para toda aquella información que no puede ser procesada o analizada utilizando procesos o herramientas tradicionales es por ello que muchas aplicaciones utilizan Bases de Datos No Relacionales para almacenar la información.

Existen plataformas como Hadoop (2009), es un marco de referencia que permite el procesamiento distribuido de grandes conjuntos de datos a través de grupos de computadoras que utilizan modelos de programación simples. Es decir que no requiere que la arquitectura o hardware sea muy costosa o de última generación. En cuanto a las bases de datos no relacionales Hadoop se integra perfectamente con Cassandra (2008) la cual es una distribución de un almacén de datos clave-valor altamente escalable y estructurada. Entre las dos aplicaciones dan soporte y crecimiento horizontal a las necesidades de las aplicaciones distribuidas como es el BigData para tener disponible la información a diferentes empresas tales como Facebook entre otras.

Arquitectura de Software

La Arquitectura de Software se refiere a la forma de cómo se estructura el diseño de un sistema, este se crea en etapas tempranas del desarrollo. La finalidad de estructurar los componentes o módulos así como el diseño del sistema tiene los propósitos de: satisfacer atributos de calidad en cuanto a desempeño, seguridad, mantenibilidad, y servir como guía en el desarrollo del proyecto. El objetivo de la arquitectura de Software consiste en desarrollar sistemas de software grandes de forma eficiente, estructurada y con capacidad de reutilización.

La Arquitectura de Software es una de las áreas los Ingenieros de Software integran en los diseños de software que realizan. Esto se debe a que la experiencia y el tiempo que tienen desarrollado proyectos les demandan tener un diseño arquitectónico de referencia. En la figura siguiente se observa los pasos a seguir para definir una Arquitectura de Software.



Determinar los requerimientos de la Arquitectura de Software: involucra crear un modelo desde el levantamiento de requerimientos que guiaran el diseño de la arquitectura basado en atributos de calidad esperado.

Diseño de la Arquitectura de Software: definir la estructura y las responsabilidades de los componentes que comprenderán la Arquitectura.

Validación de la Arquitectura de Software: básicamente se prueba la arquitectura pasando a través del diseño contra los requerimientos actuales y cualquier requerimiento futuro.

En la medida que los sistemas de software crecen en complejidad, bien sea por número de requerimientos o por el impacto de los mismos, se hace necesario establecer medios para el manejo de esta complejidad. Las Arquitecturas de Software más comunes van desde Aplicaciones Monolíticas, Arquitectura Cliente-Servidor, Arquitectura Cliente-Servidor Mejorada, Arquitectura de 3 niveles, Arquitectura de N-niveles y Arquitectura Orientada a Servicios (SOA).

Los Ingenieros de Software que comienzan a desarrollar plataformas tecnológicas robustas y escalables necesitan implementar diferentes técnicas para mejorar su codificación. Una de ellas es la identificación de problemas repetitivos en la programación por lo que se emplean patrones de diseño de software para dar solución a los problemas que se necesitan solventar.

Patrones de Diseño

Los patrones de diseño de software son una parte importante en el diseño de la Arquitectura y en el Desarrollo de software en específico de la codificación de las aplicaciones. Estos patrones tienen su origen en el diseño de sistemas con el paradigma de la Programación Orientada a Objetos y a problemas que estos no pueden resolver y que son muy recurrentes en la programación de software.

Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí, adaptada para resolver un problema de diseño general en un contexto particular.

Los patrones de diseño ayudan a mantener un código reutilizable y tener mayor control sobre los problemas que son recurrentes, de ahí que sean una solución factible para su implementación en múltiples aplicaciones.

Estos patrones son parte fundamental para el desarrollo de soluciones modulares, mantenibles y escalables. Los patrones de diseño se integran junto al modelado de sistemas con UML para identificar claramente los patrones empleados en cada componente, módulo o clase del sistema que se desarrolle. Este modelado forma parte de la Arquitectura de Software.

En la actualidad se tienen 3 categorías de patrones de diseño que se aplican dependiendo de los problemas identificados en el desarrollo de software: patrones creacionales, patrones estructurales y patrones de comportamiento.

Desarrollo de Software Orientado a Servicios (SOA)

Este tipo de Arquitectura de Software, es una de las arquitecturas más usadas para el desarrollo de soluciones de software robustas, escalables y de alto desempeño.

Las aplicaciones orientadas a servicios permiten a una aplicación ofrecer su funcionalidad como un conjunto de servicios para que sean consumidos por clientes y se especializan en proporcionar un esquema basado en mensajes de nivel de aplicación. Las características principales de este tipo de aplicaciones es su alta disponibilidad debido en parte a que cada servicio es autónomo y no afecta a la aplicación en su totalidad si llegará a fallar. Los clientes y servicios son autónomos y pueden ser consumidos de manera remota a través de la red. El mantenimiento de sistemas robustos no es complicado ya que se actualizan los servicios o se crean nuevos sin afectar directamente a los demás o la aplicación cliente que los consuma. Según Somerville existen tres (3) estándares fundamentales para la comunicación o publicación de servicios Web los cuales son:

- SOAP (del Ingles Simple Object Access Protocol): este estándar define una organización para el intercambio de datos estructurados entre servicios Web.
- WSDL (del Ingles Web Services Description Language): este protocolo define cómo representar las interfaces de servicios Web.
- UDDI(del Ingles Universal Description, Discovery and Integration): este estándar de búsqueda define como se puede organizar la información de descripción de servicios, usada por los solicitantes de los servicios para encontrar servicios.

Para los Ingenieros de Software es importante tener las habilidades necesarias para desarrollar plataformas tecnológicas que implementen la Arquitectura Orientada a Servicios (SOA) que tiene una gran penetración en el mercado de las aplicaciones empresariales, por sus bondades y características de alta disponibilidad.

Calidad en el Desarrollo de Software y Desarrollo Orientado a Pruebas

Uno de los propósitos de la Ingeniería de Software son el aseguramiento de la calidad de los productos Esto lo logramos diseñando los sistemas en un enfoque modular; documentando el software con las herramientas adecuadas y por último probar, mantener y auditar el Software.

Una de las técnicas más utilizadas e implementadas en los últimos años es el desarrollo de Software dirigido a pruebas (TDD). La técnica propone disminuir los problemas relacionados con el desarrollo de software tradicional en especial a los métodos tradicionales como el de cascada. Esta técnica se enfoca en 3 puntos principales:

- Implementación de funciones justas que el cliente necesita y no más.
- Minimización de número de defectos que llegan al Software en fase de producción.

- Desarrollar Software modular, altamente reutilizable y preparado para las adecuaciones.

El desarrollo de Software Dirigido por Pruebas es un nuevo paradigma que ofrece la facilidad de pensar en las Pruebas de Software antes que en la codificación, logrando con minimizar lo más posible los errores generados en etapas de desarrollo.

La calidad en el Desarrollo de Software esta intrínsecamente relacionado con la calidad en el proceso de Software para crearlo. A través de la implantación de procesos de desarrollo probados, bien documentados e institucionalizados, una compañía es capaz de incrementar la precisión de sus planificaciones, dando la posibilidad de establecer compromisos con sus clientes que antes no podía o no era capaz de asumir.

Modelos y Metodologías de Desarrollo de Software

Al hablar de la implementación de un modelo o una metodología para el desarrollo de software estamos haciendo referencia a la calidad de los productos de software. La obtención de un Software con calidad, implica la utilización de metodologías o procedimientos estándares para el análisis, diseño, programación y prueba del software que permitan unificar el trabajo, tanto para la labor de desarrollo como para el control de la calidad del software. Esto se debe a que los modelos tienen definidas las áreas de proceso a realizar para llevar a cabo la gestión de proyectos de software con los diferentes roles que intervienen. Existen multitud de modelos para la gestión de la calidad del software y otros sistemas y normas de gestión que se han aplicado sobre estos procesos para su evaluación.

El objetivo es crear sistemas mas confiables y de mayor seguridad al menor costo e inversión de tiempos que resuelvan, rápidamente los problemas, y que, evidentemente, a lo largo del tiempo, serán muchos mas complejos.
Las metodologías que se iran sustentando en el próximo lustro serán, fundamentalmente, las metodologías agiles, como ser: XP, Crystal, desarrollo de software adaptable (highsmith), Scrum, desarrollado manejado por rasgo (aspecto), DSDM, etc.

Corolario

La evolución de nuevas plataformas tecnológicas brinda una oportunidad importante para que diferentes áreas de las Tecnologías de la Información converjan para un fin común que es desarrollar una solución tecnológica robusta, escalable, mantenibles y sobre todo empleando calidad en su proceso desarrollo que se manifiesta en la calidad del producto desarrollado.

Es por ello que los ingenieros, desarrolladores y especialistas en Tecnologías de la Información y Desarrollo de Software deben estar en constante actualización para solventar las necesidades de las empresas, organizaciones o entidades que lo soliciten.