1. Análisis de Sistemas de Información.

1.1. Proceso de Desarrollo de Sistemas de Información.

La ingeniería es la aplicación de conocimiento científico para resolver problemas, de significado práctico inmediato, que tienen requerimientos y restricciones conflictivas.

Retomando conceptos de Sistemas y Procesos de Negocio, la Ingeniería de Sistemas es la aplicación sistemática de técnicas y métodos basadas en modelos para:

- Transformar una necesidad o carencia en el diseño y construcción de un sistema artificial (dispositivo, mecanismo o artefacto) a través del uso de un proceso iterativo de definición.
- Resolver restricciones técnicas y garantizar la compatibilidad de las interrelaciones físicas y funcionales en la concepción y diseño del sistema.
- Integrar aspectos de construcción, mantenimiento, seguridad, sustentabilidad, etc., a fin de cumplir los objetivos de costo, performance y valor del sistema.

En la Figura 1 se ilustra un marco de referencia que describe el proceso para la obtención de un sistema artificial. En la figura se ilustran distintas actividades, tales como análisis, diseño, construcción, prueba, integración, uso y mantenimiento, y obsolescencia y retirada. Estas actividades están estructuradas según un orden y en un marco de proceso evolutivo e iterativo.

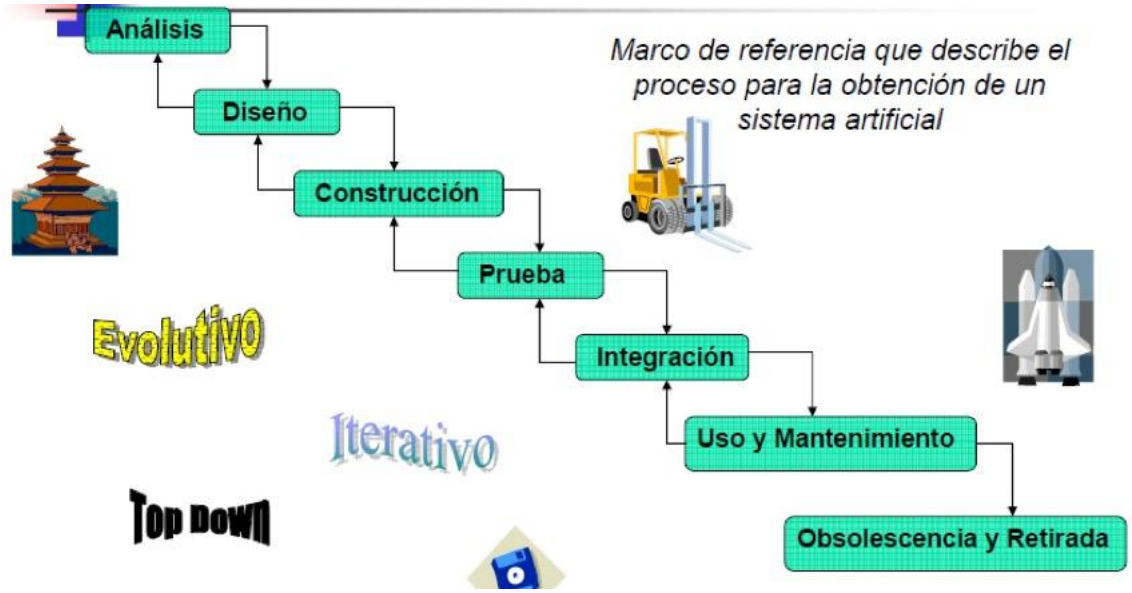

Figura 1. Marco de referencia que describe el proceso para la obtención de un sistema artificial. Obtenido de material de cátedra Sistemas y Procesos de Negocio.

En este proceso, el analista de sistemas se debe concentrar en **qué** sistema construirá, **qué** funcionalidades son requeridas, es decir, debe describir los requerimientos del sistema.

Según los conceptos visto en Sistemas y Procesos de Negocio, los objetivos de la etapa de análisis son:

- Describir los requerimientos de uso a cumplir por el sistema.
- Establecer la base para el diseño conceptual y detallado del sistema.
- Definir un conjunto de medidas de eficiencia y eficacia del sistema.

Para poder lograr estos objetivos realiza una serie de actividades empleando diversas herramientas. Entre las actividades a desarrollar en la etapa de análisis han visto que:

- Identificar las necesidades
- Estudio de factibilidad
- Determinar requisitos funcionales
- Determinar requisitos no funcionales
- Revisión

Para realizar estas actividades es posible emplear las siguientes herramientas:

- Entrevistas, encuestas, observación personal, estudio de mercado, búsqueda de antecedentes
- Modelos, y más modelos

Estas herramientas pueden variar en nombre de acuerdo con la disciplina a la cual pertenece el sistema en desarrollo, pero un factor común es la construcción de modelos. Cuando analizamos sistemas, creamos modelos del área de aplicación que nos interesa. Un modelo es una abstracción del sistema a estudiar, por lo que resulta mucho más sencillo que la realidad. Se construye con un cierto propósito, es manejable por nosotros y nos permite comprender sus propiedades, idear sistemas o rediseñar partes de este.

En este curso estudiaremos el análisis de sistemas de información. Una definición altamente aceptada de **sistema de información** (SI) es un conjunto de elementos o componentes interrelacionados para capturar, procesar, almacenar, y distribuir la información en un sistema para permitir la toma de decisiones, la gestión del sistema, y el aprendizaje colectivo. Cuenta con un mecanismo de retroalimentación para el cumplimiento de un objetivo (Stair y Reynolds, 2010). Olivé (2007) enfatiza que un SI es un sistema diseñado que captura, almacena, procesa y distribuye información sobre el estado de un dominio, por lo que los **modelos** que construiremos y analizaremos serán para **representar** ese **estado del dominio**, las posibles **consultas** que se puedan realizar sobre el mismo, y las distintas **operaciones de cambio** que se puedan aplicar sobre él.

Ahora bien, la captura, procesamiento, almacenamiento y distribución de la información se basa generalmente en componentes de software, siendo este el elemento transformador de la información. Según el glosario de la IEEE (1990), software incluye “Programas de computadoras, procedimientos, y posible documentación asociada y datos pertinentes a la operación de un sistema de computadora”. Una definición similar provee Pressman (2010): Software está compuesto de:

- Las instrucciones (programas) que al ejecutarse proporcionan las características, funciones y el grado de desempeño deseados.
- Las estructuras de datos que le permiten manipular información.
- Los documentos que describen su operación y uso.

Sommerville (2011) destaca que muchos suponen que el software es tan sólo otra palabra para los programas de cómputo. No obstante, cuando se habla de ingeniería de software, esto no sólo se refiere a los programas en sí, sino también a toda la documentación asociada y los datos de configuración requeridos para hacer que estos programas operen de manera correcta. Un sistema de software desarrollado profesionalmente es usualmente más que un solo programa. El sistema por lo regular consta de un número de programas separados y archivos de configuración que se usan para instalar dichos programas. Puede incluir documentación del sistema, que describe la estructura del sistema; documentación del usuario, que explica cómo

usar el sistema, y los sitios web para que los usuarios descarguen información reciente del producto.

Ésta es una de las principales diferencias entre el desarrollo de software profesional y el de aficionado (Sommerville, 2011). El texto que sigue a continuación y en las siguientes secciones de este documento fue extraído de la Sección 1.1 de Sommerville (2011), donde se destaca que si usted diseña un programa personal, nadie más lo usará ni tendrá que preocuparse por elaborar guías del programa, documentar el diseño del programa, etcétera. Por el contrario, si crea software que otros usarán y otros ingenieros cambiarán, entonces, en general debe ofrecer información adicional, así como el código del programa.

Hay muchos tipos diferentes de sistemas de software, desde los simples sistemas embebidos, hasta los complejos sistemas de información mundial. No tiene sentido buscar notaciones, métodos o técnicas universales para la ingeniería de software, ya que diferentes tipos de software requieren distintos enfoques. Desarrollar un sistema organizacional de información es completamente diferente de un controlador para un instrumento científico. Ninguno de estos sistemas tiene mucho en común con un juego por computadora de gráficos intensivos. Aunque todas estas aplicaciones necesitan ingeniería de software, no todas requieren las mismas técnicas de ingeniería de software.

Los ingenieros de software están interesados por el desarrollo de productos de software (es decir, software que puede venderse a un cliente). Existen dos tipos de productos de software, productos genéricos o hechos a medidas:

- Productos genéricos, consisten en sistemas independientes que se producen por una organización de desarrollo y se venden en el mercado abierto a cualquier cliente que desee comprarlos. Ejemplos de este tipo de productos incluyen software para PC, como bases de datos, procesadores de texto, paquetes de dibujo y herramientas de administración de proyectos. También abarcan las llamadas aplicaciones verticales diseñadas para cierto propósito específico, tales como sistemas de información de librería, sistemas de contabilidad o sistemas para mantener registros dentales. En este tipo de producto la organización que desarrolla el software controla la especificación del software.
- Productos hechos a medida, son sistemas que están destinados para un cliente en particular. Un contratista de software desarrolla el programa especialmente para dicho cliente. Ejemplos de este tipo de software incluyen los sistemas de control para dispositivos electrónicos, sistemas escritos para apoyar cierto proceso empresarial y los sistemas de control de tráfico aéreo. En este tipo de producto la organización que compra el software controla la especificación del software.

Sin embargo, la distinción entre estos tipos de producto de sistemas se vuelve cada vez más difusa. Ahora, cada vez más sistemas se construyen con un producto genérico como base, que luego se adapta para ajustarse a los requerimientos de un cliente. Los sistemas Enterprise Resource Planning (ERP, planeación de recursos empresariales), como el sistema SAP (https://www.sap.com/latinamerica/index.html), son los mejores ejemplos de este enfoque. Aquí, un sistema grande y complejo se adapta a una compañía al incorporar la información acerca de las reglas y los procesos empresariales, los reportes requeridos, etcétera.

El software además posee ciertos atributos ligados a características no funcionales. Diversos autores definen a estos atributos como atributos de calidad. Estos atributos no están directamente relacionados con lo que el software hace. Se refieren al comportamiento del

software mientras está ejecutando y a la estructura y organización de los programas y documentación que lo componen. Ejemplos de estos atributos son:

- Mantenibilidad. Debe ser posible que el software evolucione y que siga cumpliendo con sus especificaciones.
- Confiabilidad. Que el software funcione libre de fallas en un entorno determinado y durante un tiempo específico.
- Eficiencia. El software no debe desperdiciar los recursos del sistema. Debe usarlos adecuadamente (memoria, tiempos de respuesta, etc. …).
- Usabilidad. El software debe contar con una interfaz de usuario y documentación adecuada.

La importancia relativa de las características depende del tipo de producto y del ambiente en el que será utilizado. En algunos casos, algunos atributos pueden dominar. Por ejemplo, en sistemas de seguridad críticos de tiempo real, los atributos clave pueden ser la confiabilidad y la eficiencia, en un sistema bancario, el atributo relevante puede ser seguridad, y en un juego interactivo, el atributo principal puede ser eficiencia (tiempo de respuesta).

En el desarrollo de software se debe considerar las características del software, dado que no son las mismas que la de un producto físico convencional. Los sistemas de software son abstractos e intangibles. No están restringidos por las propiedades de los materiales, regidos por leyes físicas ni por procesos de fabricación. Esto simplifica la ingeniería de software, pues no existen límites naturales a su potencial. Sin embargo, debido a la falta de restricciones físicas, los sistemas de software pueden volverse rápidamente muy complejos, difíciles de entender y costosos de cambiar.

El software se desarrolla, no se fabrica/manufactura en el sentido clásico, de esta manera los costos del software se concentran en la ingeniería. Asimismo, el software no se desgasta, pero sí se “deteriora” (Pressman, 2010). El software no es susceptible a los problemas ambientales que hacen que el hardware o un elemento mecánico se desgaste. Por tanto, en teoría, la curva de la tasa de fallas adopta la forma de la “curva idealizada” que se aprecia en la Figura 2. Los defectos ocultos ocasionarán tasas elevadas de fallas al comienzo de la vida de un programa. Sin embargo, éstas se corrigen y la curva se aplana. La curva idealizada es una gran simplificación de los modelos reales de las fallas del software. En la Figura 2 se ilustra la curva real. Durante su vida, el software sufrirá cambios. Es probable que cuando éstos se realicen, se introduzcan errores que ocasionen que la curva de tasa de fallas tenga aumentos súbitos, como se ilustra en la “curva real”. Antes de que la curva vuelva a su tasa de fallas original de estado estable, surge la solicitud de otro cambio que hace que la curva se dispare otra vez. Poco a poco, el nivel mínimo de la tasa de fallas comienza a aumentar: el software se está deteriorando como consecuencia del cambio.

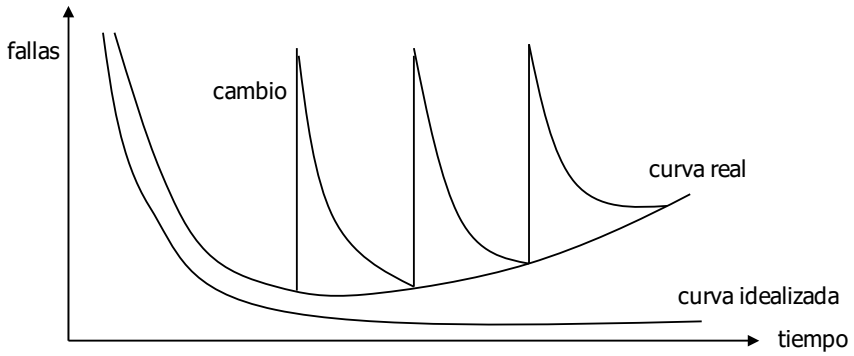

Figura 2. Curva idealizada y real de fallas en el software. Adaptada de Pressman (2010).

Mitos del Software

Pressman (2010) recopila un conjunto de mitos del software, creencias erróneas sobre éste y sobre el proceso que se utiliza para su desarrollo.

Mitos de la administración. Los gerentes que tienen responsabilidades en el software, como los de otras disciplinas, con frecuencia se hallan bajo presión para cumplir el presupuesto, mantener la programación de actividades sin desvíos y mejorar la calidad.

- Mito: Tenemos un libro lleno de estándares y procedimientos para elaborar software. ¿No le dará a mi personal todo lo que necesita saber?
    - Realidad: Tal vez exista el libro de estándares, pero ¿se utiliza? ¿Saben de su existencia los trabajadores del software? ¿Refleja la práctica moderna de la ingeniería de software? ¿Es completo? ¿Es adaptable? ¿Está dirigido a mejorar la entrega a tiempo y también se centra en la calidad? En muchos casos, la respuesta a todas estas preguntas es “no”.
- Mito: Si nos atrasamos, podemos agregar más programadores y ponernos al corriente.
    - Realidad: El desarrollo del software no es un proceso mecánico similar a la manufactura. En palabras de Brooks: “agregar personal a un proyecto de software atrasado lo atrasará más”. Al principio, esta afirmación parece ir contra la intuición. Sin embargo, a medida que se agregan personas, las que ya se encontraban trabajando deben dedicar tiempo para enseñar a los recién llegados, lo que disminuye la cantidad de tiempo dedicada al esfuerzo de desarrollo productivo. Pueden agregarse individuos, pero sólo en forma planeada y bien coordinada.
- Mito: Si decido subcontratar el proyecto de software a un tercero, puedo descansar y dejar que esa compañía lo elabore.
    - Realidad: Si una organización no comprende cómo administrar y controlar proyectos de software internamente, de manera invariable tendrá dificultades cuando subcontrate proyectos de software.

Mitos del cliente. El cliente que requiere software de computadora puede ser la persona en el escritorio de al lado, un grupo técnico en el piso inferior, el departamento de mercadotecnia y ventas, o una compañía externa que solicita software mediante un contrato. En muchos casos, el cliente sostiene mitos sobre el software porque los gerentes y profesionales de éste hacen poco para corregir la mala información. Los mitos generan falsas expectativas (por parte del cliente) y, en última instancia, la insatisfacción con el desarrollador.

- Mito: Para comenzar a escribir programas, es suficiente el enunciado general de los objetivos —podremos entrar en detalles más adelante.

    - Realidad: Aunque no siempre es posible tener el enunciado exhaustivo y estable de los requerimientos, un "planteamiento de objetivos" ambiguo es una receta para el desastre. Los requerimientos que no son ambiguos (que por lo general se obtienen en forma iterativa) se desarrollan sólo por medio de una comunicación eficaz y continua entre el cliente y el desarrollador.
- Mito: Los requerimientos del software cambian continuamente, pero el cambio se asimila con facilidad debido a que el software es flexible.
    - Realidad: Es verdad que los requerimientos del software cambian, pero el efecto que los cambios tienen varía según la época en la que se introducen. Cuando se solicitan al principio cambios en los requerimientos (antes de que haya comenzado el diseño o la elaboración de código), el efecto sobre el costo es relativamente pequeño. Sin embargo, conforme pasa el tiempo, el costo aumenta con rapidez: los recursos ya se han comprometido, se ha establecido la estructura del diseño y el cambio ocasiona perturbaciones que exigen recursos adicionales y modificaciones importantes del diseño.

Mitos del profesional. Los mitos que aún sostienen los trabajadores del software han sido alimentados por más de 50 años de cultura de programación. Durante los primeros días, la programación se veía como una forma del arte. Es difícil que mueran los hábitos y actitudes arraigados.

- Mito: Una vez que escribimos el programa y hacemos que funcione, nuestro trabajo ha terminado.
    - Realidad: Alguien dijo alguna vez que "entre más pronto se comience a 'escribir el código', más tiempo tomará hacer que funcione". Los datos de la industria indican que entre 60 y 80% de todo el esfuerzo dedicado al software ocurrirá después de entregarlo al cliente por primera vez.
- Mito: Hasta que no se haga "correr" el programa, no hay manera de evaluar su calidad.
    - Realidad: Uno de los mecanismos más eficaces de asegurar la calidad del software puede aplicarse desde la concepción del proyecto: la revisión técnica. Las revisiones del software son un "filtro de la calidad" que se ha revelado más eficaz que las pruebas para encontrar ciertas clases de defectos de software.
- Mito: El único producto del trabajo que se entrega en un proyecto exitoso es el programa que funciona.
    - Realidad: Un programa que funciona sólo es una parte de una configuración de software que incluye muchos elementos. Son varios los productos terminados (modelos, documentos, planes) que proporcionan la base de la ingeniería exitosa y, lo más importante, que guían el apoyo para el software.
- Mito: La ingeniería de software hará que generemos documentación voluminosa e innecesaria, e invariablemente nos retrasará.
    - Realidad: La ingeniería de software no consiste en producir documentos. Se trata de crear un producto de calidad. La mejor calidad conduce a menos repeticiones, lo que da como resultado tiempos de entrega más cortos.

Muchos profesionales del software reconocen la falacia de los mitos mencionados. Es lamentable que las actitudes y métodos habituales nutran la administración y las prácticas técnicas deficientes, aun cuando la realidad dicta un enfoque mejor. El primer paso hacia la formulación de soluciones prácticas para la ingeniería de software es el reconocimiento de las realidades en este campo.

1.2. Ingeniería de Software

La ingeniería de software es una disciplina de ingeniería que se interesa por todos los aspectos de la producción de software, desde las primeras etapas de la especificación del sistema hasta el mantenimiento del sistema después de que se pone en operación. En esta definición se presentan dos frases clave:

- Disciplina de ingeniería. Los ingenieros hacen que las cosas funcionen. Aplican teorías, métodos y herramientas donde es adecuado. Sin embargo, los usan de manera selectiva y siempre tratan de encontrar soluciones a problemas, incluso cuando no hay teorías ni métodos aplicables. Los ingenieros también reconocen que deben trabajar ante restricciones organizacionales y financieras, de modo que buscan soluciones dentro de tales limitaciones.
- Todos los aspectos de la producción del software. La ingeniería de software no sólo se interesa por los procesos técnicos del desarrollo de software, sino también incluye actividades como la administración del proyecto de software y el desarrollo de herramientas, así como métodos y teorías para apoyar la producción de software.

La ingeniería busca obtener resultados de la calidad requerida dentro de la fecha y del presupuesto. A menudo esto requiere contraer compromisos: los ingenieros no deben ser perfeccionistas. Sin embargo, las personas que diseñan programas para sí mismas podrían pasar tanto tiempo como deseen en el desarrollo del programa.

En general, los ingenieros de software adoptan en su trabajo un enfoque sistemático y organizado, pues usualmente ésta es la forma más efectiva de producir software de alta calidad. No obstante, la ingeniería busca seleccionar el método más adecuado para un conjunto de circunstancias y, de esta manera, un acercamiento al desarrollo más creativo y menos formal sería efectivo en ciertas situaciones. El desarrollo menos formal es particularmente adecuado para la creación de sistemas basados en la Web, que requieren una mezcla de habilidades de software y diseño gráfico.

El enfoque sistemático que se usa en la ingeniería de software se conoce en ocasiones como proceso de software. Un proceso de software es una secuencia de actividades que conducen a la elaboración de un producto de software. Existen cuatro actividades fundamentales que son comunes a todos los procesos de software, y éstas son:

1. **Especificación del software**, donde clientes e ingenieros definen el software que se producirá y las restricciones en su operación.
2. **Desarrollo del software**, donde se diseña y programa el software.
3. **Validación del software**, donde se verifica el software para asegurar que sea lo que el cliente requiere.
4. **Evolución del software**, donde se modifica el software para reflejar los requerimientos cambiantes del cliente y del mercado.

Diferentes tipos de sistemas necesitan distintos procesos de desarrollo. Por ejemplo, el software en tiempo real en una aeronave debe especificarse por completo antes de comenzar el desarrollo. En los sistemas de comercio electrónico, la especificación y el programa por lo general se desarrollan en conjunto. En consecuencia, tales actividades genéricas pueden organizarse en diferentes formas y describirse en distintos niveles de detalle, dependiendo del tipo de software que se vaya a desarrollar.

La ingeniería de software se relaciona con las ciencias de la computación y la ingeniería de sistemas:

1. Las ciencias de la computación se interesan por las teorías y los métodos que subyacen en las computadoras y los sistemas de software, en tanto que la ingeniería de software se preocupa por los asuntos prácticos de la producción del software. Cierto conocimiento de ciencias de la computación es esencial para los ingenieros de software, del mismo modo que cierto conocimiento de física lo es para los ingenieros electricistas. Sin embargo, con frecuencia la teoría de las ciencias de la computación es más aplicable a programas relativamente pequeños. Las teorías de las ciencias de la computación no siempre pueden aplicarse a grandes problemas complejos que requieren una solución de software.
2. La ingeniería de sistemas se interesa por todos los aspectos del desarrollo y la evolución de complejos sistemas, donde el software tiene un papel principal. Por lo tanto, la ingeniería de sistemas se preocupa por el desarrollo de hardware, el diseño de políticas y procesos, la implementación del sistema, así como por la ingeniería de software. Los ingenieros de sistemas intervienen en la especificación del sistema, definiendo su arquitectura global y, luego, integrando las diferentes partes para crear el sistema terminado. Están menos preocupados por la ingeniería de los componentes del sistema (hardware, software, etcétera).

La ingeniería de software es un enfoque sistemático para la producción de software que toma en cuenta los temas prácticos de costo, fecha y confiabilidad, así como las necesidades de clientes y fabricantes de software. Como este enfoque sistemático realmente implementado varía de manera drástica dependiendo de la organización que desarrolla el software, el tipo de software y los individuos que intervienen en el proceso de desarrollo, no existen métodos y técnicas universales de ingeniería de software que sean adecuados para todos los sistemas y las compañías. Más bien, durante los últimos 50 años evolucionó un conjunto de métodos y herramientas de ingeniería de software.

Quizás el factor más significativo en la determinación de qué métodos y técnicas de la ingeniería de software son más importantes, es el tipo de aplicación que está siendo desarrollada. Existen muchos diferentes tipos de aplicación, incluidos los siguientes:

1. Aplicaciones independientes. Se trata de sistemas de aplicación que corren en una computadora local, como una PC, e incluyen toda la funcionalidad necesaria y no requieren conectarse a una red. Ejemplos de tales aplicaciones son las de oficina en una PC, programas CAD, software de manipulación de fotografías, etcétera.
2. Aplicaciones interactivas basadas en transacción. Consisten en aplicaciones que se ejecutan en una computadora remota y a las que los usuarios acceden desde sus propias PC o terminales. Evidentemente, en ellas se incluyen aplicaciones Web como las de comercio electrónico, donde es posible interactuar con un sistema remoto para comprar bienes y servicios. Esta clase de aplicación también incluye sistemas empresariales, donde una organización brinda acceso a sus sistemas a través de un navegador Web o un programa de cliente de propósito específico y servicios basados en nube, como correo electrónico y compartición de fotografías. Las aplicaciones interactivas incorporan con frecuencia un gran almacén de datos al que se accede y actualiza en cada transacción.
3. Sistemas de control embebido. Se trata de sistemas de control de software que regulan y gestionan dispositivos de hardware. Numéricamente, quizás existen más sistemas embebidos que cualquier otro tipo de sistema. Algunos ejemplos de sistemas embebidos incluyen el software en un teléfono móvil (celular), el software que controla

los frenos antibloqueo de un automóvil y el software en un horno de microondas para controlar el proceso de cocinado.

4. Sistemas de procesamiento en lotes. Son sistemas empresariales que se diseñan para procesar datos en grandes lotes (batch). Procesan gran cantidad de entradas individuales para crear salidas correspondientes. Los ejemplos de sistemas batch incluyen sistemas de facturación periódica, como los sistemas de facturación telefónica y los sistemas de pago de salario.
5. Sistemas de entretenimiento. Son sistemas para uso sobre todo personal, que tienen la intención de entretener al usuario. La mayoría de estos sistemas son juegos de uno u otro tipo. La calidad de interacción ofrecida al usuario es la característica más importante de los sistemas de entretenimiento.
6. Sistemas para modelado y simulación. Éstos son sistemas que desarrollan científicos e ingenieros para modelar procesos o situaciones físicas, que incluyen muchos objetos separados interactuantes. Dichos sistemas a menudo son computacionalmente intensivos y para su ejecución requieren sistemas paralelos de alto desempeño.
7. Sistemas de adquisición de datos. Son sistemas que desde su entorno recopilan datos usando un conjunto de sensores, y envían dichos datos para su procesamiento a otros sistemas. El software tiene que interactuar con los sensores y se instala regularmente en un ambiente hostil, como en el interior de un motor o en una ubicación remota.
8. Sistemas de sistemas. Son sistemas compuestos de un cierto número de sistemas de software. Algunos de ellos son producto del software genérico, como un programa de hoja de cálculo. Otros sistemas en el ensamble pueden estar especialmente escritos para ese entorno.

Desde luego, los límites entre estos tipos de sistemas son difusos, incluso, en función del criterio de clasificación, existen otras clasificaciones posibles. Pressman (2010) los clasifica según su existencia previa como software nuevo, heredado, o de integración.

Para cada tipo de sistema se usan distintas técnicas de ingeniería de software, porque el software tiene características muy diferentes. Por ejemplo, un sistema de control embebido en un automóvil es crítico para la seguridad y se graba en la ROM cuando se instala en el vehículo; por consiguiente, es muy costoso cambiarlo. Tal sistema necesita verificación y validación muy exhaustivas, de tal modo que se minimicen las probabilidades de volver a llamar para revisión a automóviles, después de su venta, para corregir los problemas del software. La interacción del usuario es mínima (o quizás inexistente), por lo que no hay necesidad de usar un proceso de desarrollo que se apoye en el prototipo de interfaz de usuario.

Para un sistema basado en la Web sería adecuado un enfoque basado en el desarrollo y la entrega iterativos, con un sistema de componentes reutilizables. Sin embargo, tal enfoque podría no ser práctico para un sistema de sistemas, donde tienen que definirse por adelantado las especificaciones detalladas de las interacciones del sistema, de modo que cada sistema se desarrolle por separado.

No obstante, existen fundamentos de ingeniería de software que se aplican a todos los tipos de sistema de software:

1. Deben llevarse a cabo usando un proceso de desarrollo administrado y comprendido. La organización que diseña el software necesita planear el proceso de desarrollo, así como tener ideas claras acerca de lo que producirá y el tiempo en que estará completado. Desde luego, se usan diferentes procesos para distintos tipos de software.

2. La confiabilidad y el desempeño son importantes para todos los tipos de sistemas. El software tiene que comportarse como se espera, sin fallas, y cuando se requiera estar disponible. Debe ser seguro en su operación y, tanto como sea posible, también contra ataques externos. El sistema tiene que desempeñarse de manera eficiente y no desperdiciar recursos.
3. Es importante comprender y gestionar la especificación y los requerimientos del software (lo que el software debe hacer). Debe conocerse qué esperan de él los diferentes clientes y usuarios del sistema, y gestionar sus expectativas, para entregar un sistema útil dentro de la fecha y presupuesto.
4. Tiene que usar de manera tan efectiva como sea posible los recursos existentes. Esto significa que, donde sea adecuado, hay que reutilizar el software que se haya desarrollado, en vez de diseñar uno nuevo.

La Ingeniería de software es una disciplina Ingenieril, en consecuencia, los ingenieros aplican teorías, métodos y herramientas para el desarrollo profesional de software (Pressman, 2010).

El proceso de ingeniería de software es el aglutinante que une las capas de la tecnología y permite el desarrollo racional y oportuno del software de cómputo. El proceso define una estructura que debe establecerse para la obtención eficaz de tecnología de ingeniería de software. El proceso de software forma la base para el control de la administración de proyectos de software, y establece el contexto en el que se aplican métodos técnicos, se generan productos del trabajo (modelos, documentos, datos, reportes, formatos, etc.), se establecen puntos de referencia, se asegura la calidad y se administra el cambio de manera apropiada.

Los métodos de la ingeniería de software proporcionan la experiencia técnica para elaborar software. Incluyen un conjunto amplio de tareas, como comunicación, análisis de los requerimientos, modelado del diseño, construcción del programa, pruebas y apoyo. Los métodos de la ingeniería de software se basan en un conjunto de principios fundamentales que gobiernan cada área de la tecnología e incluyen actividades de modelado y otras técnicas descriptivas.

Las herramientas de la ingeniería de software proporcionan un apoyo automatizado o semiautomatizado para el proceso y los métodos (Pressman, 2010).

1.3. Proceso de Software

Un proceso de software es una serie de actividades relacionadas que conduce a la elaboración de un producto de software. A continuación, se incluye distintas secciones del Capítulo 2, Procesos de Software, de Sommerville (2011).

Existen diferentes procesos de software, pero todos deben incluir cuatro actividades que son fundamentales para la ingeniería de software: especificación, diseño e implementación, validación y evolución del software.

En cierta forma, tales actividades forman parte de todos los procesos de software. Por supuesto, en la práctica éstas son actividades complejas en sí mismas e incluyen subactividades tales como la validación de requerimientos, el diseño arquitectónico, la prueba de unidad, etcétera. También existen actividades de soporte al proceso, como la documentación y el manejo de la configuración del software.

Cuando los procesos se discuten y describen, por lo general se habla de actividades como especificar un modelo de datos, diseñar una interfaz de usuario, etcétera, así como del orden de dichas actividades. Sin embargo, al igual que las actividades, también las descripciones de los procesos deben incluir:

1. Productos, que son los resultados de una actividad del proceso. Por ejemplo, el resultado de la actividad del diseño arquitectónico es un modelo de la arquitectura de software.
2. Roles, que reflejan las responsabilidades de la gente que interviene en el proceso. Ejemplos de roles: gerente de proyecto, gerente de configuración, programador, etcétera.
3. Precondiciones y postcondiciones, que son declaraciones válidas antes y después de que se realice una actividad del proceso o se cree un producto. Por ejemplo, antes de comenzar el diseño arquitectónico, una precondición es que el cliente haya aprobado todos los requerimientos; después de terminar esta actividad, una postcondición podría ser que se revisen aquellos modelos UML que describen la arquitectura.

Los procesos de software son complejos y, como todos los procesos intelectuales y creativos, se apoyan en personas con capacidad de juzgar y tomar decisiones. No hay un proceso ideal; además, la mayoría de las organizaciones han diseñado sus propios procesos de desarrollo de software. Los procesos han evolucionado para beneficiarse de las capacidades de la gente en una organización y de las características específicas de los sistemas que se están desarrollando. Para algunos sistemas, como los sistemas críticos, se requiere de un proceso de desarrollo muy estructurado. Para los sistemas empresariales, con requerimientos rápidamente cambiantes, es probable que sea más efectivo un proceso menos formal y flexible.

En ocasiones, los procesos de software se clasifican como dirigidos por un plan o como procesos ágiles. Los procesos dirigidos por un plan son aquellos donde todas las actividades del proceso se planean por anticipado y el avance se mide contra dicho plan. En los procesos ágiles, la planeación es incremental y es más fácil modificar el proceso para reflejar los requerimientos cambiantes del cliente.

1.4. Modelos de Procesos de Software

La Sección 2.1 de Sommerville (2011) introduce distintos modelos de procesos de desarrollo de software de manera general, sin incluir el detalle de las actividades específicas. Tales modelos genéricos no son descripciones definitivas de los procesos de software. Más bien, son abstracciones del proceso que se utilizan para explicar los diferentes enfoques del desarrollo de software. Se pueden considerar marcos del proceso que se extienden y se adaptan para crear procesos más específicos de ingeniería de software. Estos permiten:

- Determinar el orden de las etapas involucradas en el desarrollo del software,
- establecer el criterio de transición para progresar de una etapa a la siguiente (Bohem): criterio para determinar la finalización y criterio para comenzar y elegir la siguiente.

Un modelo prescriptivo o ciclo de vida apunta a:

- ¿Qué debemos hacer a continuación?
- ¿Por cuánto tiempo debemos hacerlo?

Las principales diferencias entre distintos modelos de ciclo de vida están divididas en tres grandes visiones:

  - El **alcance** del ciclo de vida (hasta dónde se desea llegar con el proyecto): sólo saber si es viable el desarrollo de un producto, el desarrollo completo o el desarrollo completo más actualizaciones y mantenimiento.
  - La **cualidad** y **cantidad** de las etapas en que será dividido el ciclo de vida: según el que sea adoptado y el proyecto para el cual sea adoptado.

- La **estructura** y la **sucesión** de las etapas, si hay realimentación entre ellas y si hay libertad de repetirlas (iteración).

Los modelos del proceso que se examinan son:

1. **El modelo en cascada (waterfall)**. Éste toma las actividades fundamentales del proceso de especificación, desarrollo, validación y evolución y, luego, los representa como fases separadas del proceso, tal como especificación de requerimientos, diseño de software, implementación, pruebas, etcétera.
2. **Desarrollo incremental**. Este enfoque vincula las actividades de especificación, desarrollo y validación. El sistema se desarrolla como una serie de versiones (incrementos), y cada versión añade funcionalidad a la versión anterior.
3. **Ingeniería de software orientada a la reutilización**. Este enfoque se basa en la existencia de un número significativo de componentes reutilizables. El proceso de desarrollo del sistema se enfoca en la integración de estos componentes en un sistema, en vez de desarrollarlo desde cero.

Dichos modelos no son mutuamente excluyentes y con frecuencia se usan en conjunto, sobre todo para el desarrollo de grandes sistemas. Para este tipo de sistemas, tiene sentido combinar algunas de las mejores características de los modelos de desarrollo en cascada e incremental. Se necesita contar con información sobre los requerimientos esenciales del sistema para diseñar la arquitectura de software que apoye dichos requerimientos. No puede desarrollarse de manera incremental. Los subsistemas dentro de un sistema más grande se desarrollan usando diferentes enfoques. Partes del sistema que son bien comprendidas pueden especificarse y desarrollarse al utilizar un proceso basado en cascada. Partes del sistema que por adelantado son difíciles de especificar, como la interfaz de usuario, siempre deben desarrollarse con un enfoque incremental. En consecuencia, se debe escoger una estrategia de desarrollo apropiada que se selecciona de acuerdo a:

- naturaleza del proyecto y aplicación
- controles y entregas requeridas
- restricciones del contexto, organizacionales

Ninguno es mejor que otro, cada proyecto tiene el más apropiado

La ausencia de un modelo de proceso explícito es descripta por el proceso Codificar y Corregir (Code & Fix). Este modelo contiene dos pasos:

- Escribir código.
- Corregir problemas en el código.

Los problemas presentes en esta forma de trabajo se pueden resumir en:

- Después de un número de correcciones, el código puede tener una muy mala estructura, lo que hace que los arreglos sean muy costosos.
- El código no satisface las necesidades del usuario (no hay fases de requerimientos) , por lo que es rechazado o su reconstrucción es muy cara.
- El código no fue planeado para modificación, no es flexible y es difícil de corregir.

Sin embargo, sería utilizable cuando existe una programación de desarrollo ajustada, se desarrolla código rápidamente y se ven ‘resultados’ inmediatamente.

1.5. El Modelo en Cascada

El primer modelo publicado sobre el proceso de desarrollo de software se derivó a partir de procesos más generales de ingeniería de sistemas, publicado por Royce en 1970. Este modelo

se ilustra en la Figura 3. Debido al paso de una fase en cascada a otra, este modelo se conoce como “modelo en cascada” o ciclo de vida del software. El modelo en cascada es un ejemplo de un proceso dirigido por un plan; en principio, se debe planear y programar todas las actividades del proceso, antes de comenzar a trabajar con ellas.

Las principales etapas del modelo en cascada ilustradas en la Figura 3 reflejan directamente las actividades fundamentales del desarrollo:

1. **Análisis y definición de requerimientos**. Los servicios, las restricciones y las metas del sistema se establecen mediante consulta a los usuarios del sistema. Luego, se definen con detalle y sirven como una especificación del sistema.
2. **Diseño del sistema y del software**. El proceso de diseño de sistemas asigna los requerimientos, para sistemas de hardware o de software, al establecer una arquitectura de sistema global. El diseño del software implica identificar y describir las abstracciones fundamentales del sistema de software y sus relaciones.
3. **Implementación y prueba de unidad**. Durante esta etapa, el diseño de software se realiza como un conjunto de programas o unidades del programa. La prueba de unidad consiste en verificar que cada unidad cumpla con su especificación.
4. **Integración y prueba de sistema**. Las unidades del programa o los programas individuales se integran y prueban como un sistema completo para asegurarse de que se cumplan los requerimientos de software. Después de probarlo, se libera el sistema de software al cliente.
5. **Operación y mantenimiento**. Por lo general (aunque no necesariamente), ésta es la fase más larga del ciclo de vida, donde el sistema se instala y se pone en práctica. El mantenimiento incluye corregir los errores que no se detectaron en etapas anteriores del ciclo de vida, mejorar la implementación de las unidades del sistema e incrementar los servicios del sistema conforme se descubren nuevos requerimientos.

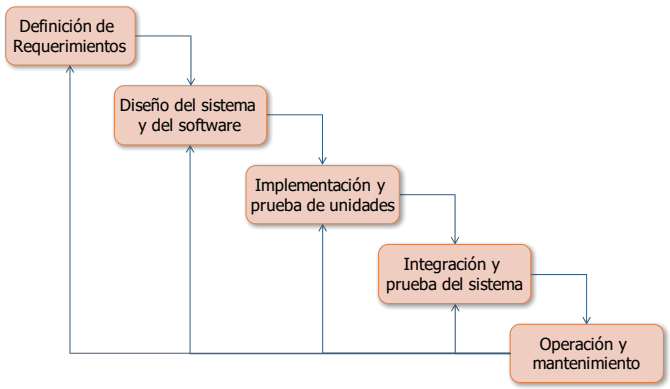

Figura 3. Modelo en Cascada

En principio, el resultado de cada fase consiste en uno o más documentos que se autorizaron (“firmaron”). La siguiente fase no debe comenzar sino hasta que termine la fase previa. En la práctica, dichas etapas se traslapan y se nutren mutuamente de información. Durante el diseño se identifican los problemas con los requerimientos. En la codificación se descubren problemas de diseño, y así sucesivamente. El proceso de software no es un simple modelo lineal, sino que implica retroalimentación de una fase a otra. Entonces, es posible que los documentos generados en cada fase deban modificarse para reflejar los cambios que se realizan.

Debido a los costos de producción y aprobación de documentos, las iteraciones suelen ser onerosas e implicar un rediseño significativo. Por lo tanto, después de un pequeño número de iteraciones, es normal detener partes del desarrollo, como la especificación, y continuar con

etapas de desarrollo posteriores. Los problemas se dejan para una resolución posterior, se ignoran o se programan. Este freno prematuro de los requerimientos quizá signifique que el sistema no hará lo que el usuario desea. También podría conducir a sistemas mal estructurados conforme los problemas de diseño se evadan con la implementación de trucos.

Durante la fase final del ciclo de vida (operación y mantenimiento), el software se pone en servicio. Se descubren los errores y las omisiones en los requerimientos originales del software. Surgen los errores de programa y diseño, y se detecta la necesidad de nueva funcionalidad. Por lo tanto, el sistema debe evolucionar para mantenerse útil. Hacer tales cambios (mantenimiento de software) puede implicar la repetición de etapas anteriores del proceso.

El modelo en cascada es consecuente con otros modelos del proceso de ingeniería y en cada fase se produce documentación. Esto hace que el proceso sea visible, de modo que los administradores monitoricen el progreso contra el plan de desarrollo. Su principal problema es la partición inflexible del proyecto en distintas etapas. Tienen que establecerse compromisos en una etapa temprana del proceso, lo que dificulta responder a los requerimientos cambiantes del cliente.

En principio, el modelo en cascada sólo debe usarse cuando los requerimientos se entiendan bien y sea improbable el cambio radical durante el desarrollo del sistema. Sin embargo, el modelo en cascada refleja el tipo de proceso utilizado en otros proyectos de ingeniería. Como es más sencillo emplear un modelo de gestión común durante todo el proyecto, aún son de uso común los procesos de software basados en el modelo en cascada.

Una variación importante del modelo en cascada es el desarrollo de sistemas formales, donde se crea un modelo matemático para una especificación del sistema. Después se corrige este modelo, mediante transformaciones matemáticas que preservan su consistencia en un código ejecutable. Con base en la suposición de que son correctas sus transformaciones matemáticas, se puede aseverar, por lo tanto, que un programa generado de esta forma es consecuente con su especificación.

Los procesos formales de desarrollo, como el que se basa en el método B son muy adecuados para el desarrollo de sistemas que cuenten con rigurosos requerimientos de seguridad, fiabilidad o protección. El enfoque formal simplifica la producción de un caso de protección o seguridad. Esto demuestra a los clientes o reguladores que el sistema en realidad cumple sus requerimientos de protección o seguridad.

Los procesos basados en transformaciones formales se usan por lo general sólo en el desarrollo de sistemas críticos para protección o seguridad. Requieren experiencia especializada. Para la mayoría de los sistemas, este proceso no ofrece costo/beneficio significativos sobre otros enfoques en el desarrollo de sistemas.

1.6. Desarrollo Incremental

El desarrollo incremental se basa en la idea de diseñar una implementación inicial, exponer ésta al comentario del usuario, y luego desarrollarla en sus diversas versiones hasta producir un sistema adecuado (Figura 4). Las actividades de especificación, desarrollo y validación están entrelazadas en vez de separadas, con rápida retroalimentación a través de las actividades.

El desarrollo de software incremental, que es una parte fundamental de los enfoques ágiles, es mejor que un enfoque en cascada para la mayoría de los sistemas empresariales, de comercio electrónico y personales. El desarrollo incremental refleja la forma en que se resuelven problemas. Rara vez se trabaja por adelantado una solución completa del problema, más bien se avanza en una serie de pasos hacia una solución y se retrocede cuando se detecta que se

cometieron errores. Al desarrollar el software de manera incremental, resulta más barato y fácil realizar cambios en el software conforme éste se diseña.

Cada incremento o versión del sistema incorpora algunas de las funciones que necesita el cliente. Por lo general, los primeros incrementos del sistema incluyen la función más importante o la más urgente. Esto significa que el cliente puede evaluar el desarrollo del sistema en una etapa relativamente temprana, para constatar si se entrega lo que se requiere. En caso contrario, sólo el incremento actual debe cambiarse y, posiblemente, definir una nueva función para incrementos posteriores.

Comparado con el modelo en cascada, el desarrollo incremental tiene tres beneficios importantes:

1. Se reduce el costo de adaptar los requerimientos cambiantes del cliente. La cantidad de análisis y la documentación que tiene que reelaborarse son mucho menores de lo requerido con el modelo en cascada.
2. Es más sencillo obtener retroalimentación del cliente sobre el trabajo de desarrollo que se realizó. Los clientes pueden comentar las demostraciones del software y darse cuenta de cuánto se ha implementado. Los clientes encuentran difícil juzgar el avance a partir de documentos de diseño de software.
3. Es posible que sea más rápida la entrega e implementación de software útil al cliente, aun si no se ha incluido toda la funcionalidad. Los clientes tienen posibilidad de usar y ganar valor del software más temprano de lo que sería posible con un proceso en cascada.

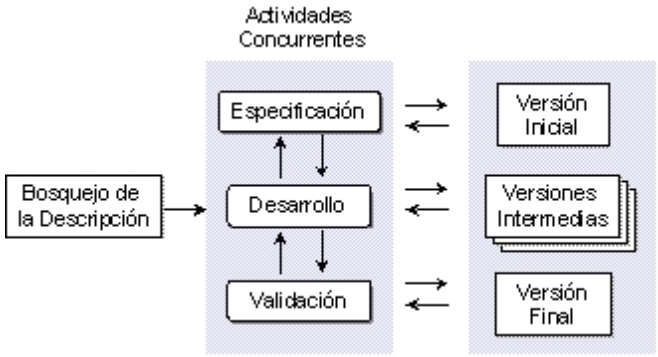

Figura 4. Desarrollo incremental

El desarrollo incremental ahora es en cierta forma el enfoque más común para el desarrollo de sistemas de aplicación. Este enfoque puede estar basado en un plan, ser ágil o, más usualmente, una mezcla de dichos enfoques. En un enfoque basado en un plan se identifican por adelantado los incrementos del sistema; si se adopta un enfoque ágil, se detectan los primeros incrementos, aunque el desarrollo de incrementos posteriores depende del avance y las prioridades del cliente.

Desde una perspectiva administrativa, el enfoque incremental tiene dos problemas:

1. El proceso no es visible. Los administradores necesitan entregas regulares para medir el avance. Si los sistemas se desarrollan rápidamente, resulta poco efectivo en términos de costos producir documentos que reflejen cada versión del sistema.
2. La estructura del sistema tiende a degradarse conforme se tienen nuevos incrementos. A menos que se gaste tiempo y dinero en la refactorización para mejorar el software, el cambio regular tiende a corromper su estructura. La incorporación de más cambios de software se vuelve cada vez más difícil y costosa.

Los problemas del desarrollo incremental se tornan particularmente agudos para sistemas grandes, complejos y de larga duración, donde diversos equipos desarrollan diferentes partes del sistema. Los grandes sistemas necesitan de un marco o una arquitectura estable y es necesario definir con claridad, respecto a dicha arquitectura, las responsabilidades de los distintos equipos que trabajan en partes del sistema. Esto debe planearse por adelantado en vez de desarrollarse de manera incremental.

Se puede desarrollar un sistema incremental y exponerlo a los clientes para su comentario, sin realmente entregarlo e implementarlo en el entorno del cliente. La entrega y la implementación incrementales significan que el software se usa en procesos operacionales reales. Esto no siempre es posible, ya que la experimentación con un nuevo software llega a alterar los procesos empresariales normales.

1.7. Ingeniería de software orientada a la reutilización

En la mayoría de los proyectos de software hay cierta reutilización de software. Sucede con frecuencia de manera informal, cuando las personas que trabajan en el proyecto conocen diseños o códigos que son similares a lo que se requiere. Los buscan, los modifican según se necesite y los incorporan en sus sistemas.

Esta reutilización informal ocurre independientemente del proceso de desarrollo que se emplee. Sin embargo, los enfoques orientados a la reutilización se apoyan en una gran base de componentes de software reutilizable y en la integración de frameworks para la composición de dichos componentes. En ocasiones, tales componentes son sistemas legados (sistemas comerciales, off-the-shelf o COTS) que pueden mejorar la funcionalidad específica, como el procesador de textos o la hoja de cálculo.

En la Figura 5 se muestra un modelo del proceso general para desarrollo basado en reutilización.

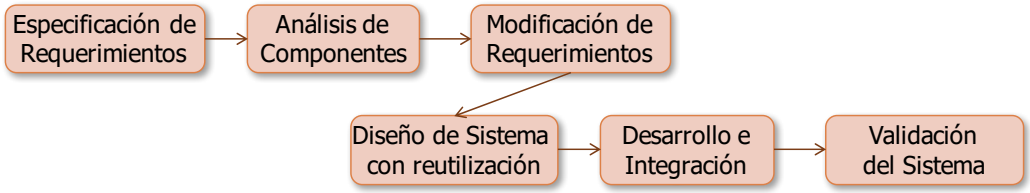

Figura 5. Modelo de proceso basado en reutilización.

Aunque la etapa inicial de especificación de requerimientos y la etapa de validación se comparan con otros procesos de software en un proceso orientado a la reutilización, las etapas intermedias son diferentes. Dichas etapas son:

1. Análisis de componentes. Dada la especificación de requerimientos, se realiza una búsqueda de componentes para implementar dicha especificación. Por lo general, no hay coincidencia exacta y los componentes que se usan proporcionan sólo parte de la funcionalidad requerida.

2. Modificación de requerimientos. Durante esta etapa se analizan los requerimientos usando información de los componentes descubiertos. Luego se modifican para reflejar los componentes disponibles. Donde las modificaciones son imposibles, puede regresarse a la actividad de análisis de componentes para buscar soluciones alternativas.

3. Diseño de sistema con reutilización. Durante esta fase se diseña el marco conceptual del sistema o se reutiliza un marco conceptual existente. Los creadores toman en cuenta los componentes que se reutilizan y organizan el marco de referencia para atenderlo. Es posible que deba diseñarse algo de software nuevo, si no están disponibles los componentes reutilizables.

4. Desarrollo e integración. Se diseña el software que no puede procurarse de manera externa, y se integran los componentes y los sistemas COTS para crear el nuevo sistema. La integración del sistema, en este modelo, puede ser parte del proceso de desarrollo, en vez de una actividad independiente.

Existen tres tipos de componentes de software que pueden usarse en un proceso orientado a la reutilización:

1. Servicios Web que se desarrollan en concordancia para atender servicios estándares y que están disponibles para la invocación remota.
2. Colecciones de objetos que se desarrollan como un paquete para su integración con un marco de componentes como .NET o JEE.
3. Sistemas de software independientes que se configuran para usar en un entorno particular.

La ingeniería de software orientada a la reutilización tiene la clara ventaja de reducir la cantidad de software a desarrollar y, por lo tanto, la de disminuir costos y riesgos; por lo general, también conduce a entregas más rápidas del software. Sin embargo, son inevitables los compromisos de requerimientos y esto conduciría hacia un sistema que no cubra las necesidades reales de los usuarios. Más aún, se pierde algo de control sobre la evolución del sistema, conforme las nuevas versiones de los componentes reutilizables no estén bajo el control de la organización que los usa.

1.8. Gestión del Cambio

La presente sección reproduce la sección 2.3 de Sommerville (2011), en tal sección se destaca que el cambio es inevitable en todos los grandes proyectos de software. Los requerimientos del sistema varían conforme la empresa procura que el sistema responda a presiones externas y se modifican las prioridades administrativas. A medida que se ponen a disposición nuevas tecnologías, surgen nuevas posibilidades de diseño e implementación. Por ende, cualquiera que sea el modelo del proceso de software utilizado, es esencial que ajuste los cambios al software a desarrollar.

El cambio se agrega a los costos del desarrollo de software debido a que, por lo general, significa que el trabajo ya terminado debe volver a realizarse. A esto se le llama **rehacer**. Por ejemplo, si se analizaron las relaciones entre los requerimientos en un sistema y se identifican nuevos requerimientos, parte o todo el análisis de requerimientos tiene que repetirse. Entonces, es necesario rediseñar el sistema para entregar los nuevos requerimientos, cambiar cualquier programa que se haya desarrollado y volver a probar el sistema. Existen dos enfoques relacionados que se usan para reducir los costos del rehacer:

1. **Evitar el cambio**, donde el proceso de software incluye actividades que anticipan cambios posibles antes de requerirse la labor significativa de rehacer. Por ejemplo, puede desarrollarse un sistema prototipo para demostrar a los clientes algunas características clave del sistema. Ellos podrán experimentar con el prototipo y refinar sus requerimientos, antes de comprometerse con mayores costos de producción de software.
2. **Tolerancia al cambio**, donde el proceso se diseña de modo que los cambios se ajusten con un costo relativamente bajo. Por lo general, esto comprende algunas formas de desarrollo incremental. Los cambios propuestos pueden implementarse en incrementos que aún no se desarrollan. Si no es posible, entonces tal vez sólo un incremento (una pequeña parte del sistema) tendría que alterarse para incorporar el cambio.

En esta sección se estudian dos formas de enfrentar el cambio y los requerimientos cambiantes del sistema. Se trata de lo siguiente:

1. Prototipo de sistema, donde rápidamente se desarrolla una versión del sistema o una parte del mismo, para comprobar los requerimientos del cliente y la factibilidad de algunas decisiones de diseño. Esto apoya el hecho de evitar el cambio, al permitir que los usuarios experimenten con el sistema antes de entregarlo y así refinar sus requerimientos. Como resultado, es probable que se reduzca el número de propuestas de cambio de requerimientos posterior a la entrega.
2. Entrega incremental, donde los incrementos del sistema se entregan al cliente para su comentario y experimentación. Esto apoya tanto al hecho de evitar el cambio como a tolerar el cambio. Por un lado, evita el compromiso prematuro con los requerimientos para todo el sistema y, por otro, permite la incorporación de cambios en incrementos mayores a costos relativamente bajos.

La noción de refactorización, esto es, el mejoramiento de la estructura y organización de un programa, es también un mecanismo importante que apoya la tolerancia al cambio.

Prototipo de sistema

Un prototipo es una versión inicial de un sistema de software que se usa para demostrar conceptos, tratar opciones de diseño y encontrar más sobre el problema y sus posibles soluciones. El rápido desarrollo iterativo del prototipo es esencial, de modo que se controlen los costos, y los interesados en el sistema experimenten por anticipado con el prototipo durante el proceso de software.

Un prototipo de software se usa en un proceso de desarrollo de software para contribuir a anticipar los cambios que se requieran:

1. En el proceso de ingeniería de requerimientos, un prototipo ayuda con la selección y validación de requerimientos del sistema.
2. En el proceso de diseño de sistemas, un prototipo sirve para buscar soluciones específicas de software y apoyar el diseño de interfaces del usuario.

Los prototipos del sistema permiten a los usuarios ver qué tan bien el sistema apoya su trabajo. Pueden obtener nuevas ideas para requerimientos y descubrir áreas de fortalezas y debilidades en el software. Entonces, proponen nuevos requerimientos del sistema. Más aún, conforme se desarrolla el prototipo, quizá se revelen errores y omisiones en los requerimientos propuestos. Una función descrita en una especificación puede parecer útil y bien definida. Sin embargo, cuando dicha función se combina con otras operaciones, los usuarios descubren frecuentemente que su visión inicial era incorrecta o estaba incompleta. Entonces, se modifica la especificación del sistema con la finalidad de reflejar su nueva comprensión de los requerimientos.

Mientras se elabora el sistema para la realización de experimentos de diseño, un prototipo del mismo sirve para comprobar la factibilidad de un diseño propuesto. Por ejemplo, puede crearse un prototipo del diseño de una base de datos y ponerse a prueba, con el objetivo de comprobar que soporta de forma eficiente el acceso de datos para las consultas más comunes del usuario. Asimismo, la creación de prototipos es una parte esencial del proceso de diseño de interfaz del usuario. Debido a la dinámica natural de las interfaces de usuario, las descripciones textuales y los diagramas no son suficientemente buenos para expresar los requerimientos de la interfaz del usuario. Por lo tanto, la creación rápida de prototipos con la participación del usuario final

es la única forma sensible para desarrollar interfaces de usuario gráficas para sistemas de software.

En la Figura 6 se muestra un modelo del proceso para desarrollo de prototipos. Los objetivos de la creación de prototipos deben estar explícitos desde el inicio del proceso. Esto tendría la finalidad de desarrollar un sistema para un prototipo de la interfaz del usuario, y diseñar un sistema que valide los requerimientos funcionales del sistema o desarrolle un sistema que demuestre a los administradores la factibilidad de la aplicación.

Cabe destacar que, si los objetivos quedan sin especificar, los administradores o usuarios finales pueden llegar a malinterpretar la función del prototipo. En consecuencia, es posible que no se obtengan los beneficios esperados del desarrollo del prototipo.

La siguiente etapa del proceso consiste en decidir qué poner y, algo quizá más importante, qué dejar fuera del sistema de prototipo. Para reducir los costos de creación de prototipos y acelerar las fechas de entrega, es posible dejar cierta funcionalidad fuera del prototipo y, también, decidir hacer más flexible los requerimientos no funcionales, como el tiempo de respuesta y la utilización de memoria. El manejo y la gestión de errores pueden ignorarse, a menos que el objetivo del prototipo sea establecer una interfaz de usuario. Además, es posible reducir los estándares de fiabilidad y calidad del programa.

La etapa final del proceso es la evaluación del prototipo. Hay que tomar provisiones durante esta etapa para la capacitación del usuario y usar los objetivos del prototipo para derivar un plan de evaluación. Los usuarios requieren tiempo para sentirse cómodos con un nuevo sistema e integrarse a un patrón normal de uso. Una vez que utilizan el sistema de manera normal, descubren errores y omisiones en los requerimientos.

Un problema general con la creación de prototipos es que quizás el prototipo no se utilice necesariamente en la misma forma que el sistema final. El revisor del prototipo tal vez no sea un usuario típico del sistema. También, podría resultar insuficiente el tiempo de capacitación durante la evaluación del prototipo.

En ocasiones, los desarrolladores están presionados por los administradores para entregar prototipos desechables, sobre todo cuando existen demoras en la entrega de la versión final del software. Sin embargo, por lo general esto no es aconsejable:

1. Puede ser imposible corregir el prototipo para cubrir requerimientos no funcionales, como los requerimientos de rendimiento, seguridad, robustez y fiabilidad, ignorados durante el desarrollo del prototipo.

2. El cambio rápido durante el desarrollo significa claramente que el prototipo no está documentado. La única especificación de diseño es el código del prototipo. Esto no es muy bueno para el mantenimiento a largo plazo.

3. Probablemente los cambios realizados durante el desarrollo de prototipos degradarán la estructura del sistema, y este último será difícil y costoso de mantener.

4. Por lo general, durante el desarrollo de prototipos se hacen más flexibles los estándares de calidad de la organización.

Los prototipos no tienen que ser ejecutables para ser útiles. Los modelos en papel de la interfaz de usuario del sistema pueden ser efectivos para ayudar a los usuarios a refinar un diseño de interfaz y trabajar a través de escenarios de uso. Su desarrollo es muy económico y suelen construirse en pocos días. Hoy en día son muy comunes las aplicaciones en la web para el desarrollo de prototipos, por ejemplo, las disponibles en: www.mockflow.com, iplotz.com, pidoco.com.

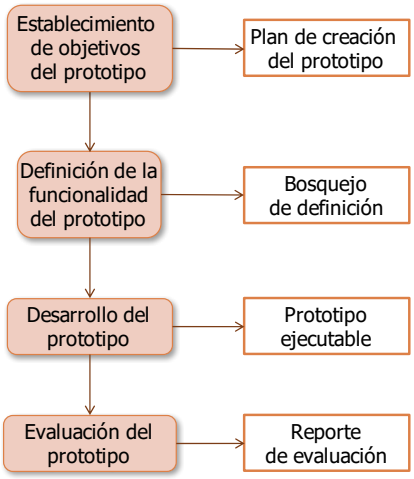

Figura 6. Proceso de desarrollo del prototipo

Entrega Incremental

La entrega incremental es un enfoque al desarrollo de software donde algunos de los incrementos diseñados se entregan al cliente y se implementan para usarse en un entorno operacional. En un proceso de entrega incremental, los clientes identifican, en un bosquejo, los servicios que proporciona el sistema. Identifican cuáles servicios son más importantes y cuáles son menos significativos para ellos. Entonces, se define un número de incrementos de entrega, y cada incremento proporciona un subconjunto de la funcionalidad del sistema. La asignación de servicios por incrementos depende de la prioridad del servicio, donde los servicios de más alta prioridad se implementan y entregan primero.

Una vez identificados los incrementos del sistema, se definen con detalle los requerimientos de los servicios que se van a entregar en el primer incremento, y se desarrolla ese incremento. Durante el desarrollo, puede haber un mayor análisis de requerimientos para incrementos posteriores, aun cuando se rechacen cambios de requerimientos para el incremento actual.

Una vez completado y entregado el incremento, los clientes lo ponen en servicio. Esto significa que toman la entrega anticipada de la funcionalidad parcial del sistema. Pueden experimentar con el sistema que les ayuda a clarificar sus requerimientos, para posteriores incrementos del sistema. A medida que se completan nuevos incrementos, se integran con los incrementos existentes, de modo que con cada incremento entregado mejore la funcionalidad del sistema.

La entrega incremental tiene algunas ventajas:

1. Los clientes pueden usar los primeros incrementos como prototipos y adquirir experiencia que informe sobre sus requerimientos, para posteriores incrementos del sistema. A diferencia de los prototipos, éstos son parte del sistema real, de manera que no hay reaprendizaje cuando está disponible el sistema completo.
2. Los clientes no deben esperar hasta la entrega completa del sistema, antes de ganar valor del mismo. El primer incremento cubre sus requerimientos más críticos, de modo que es posible usar inmediatamente el software.
3. El proceso mantiene los beneficios del desarrollo incremental en cuanto a que debe ser relativamente sencillo incorporar cambios al sistema.
4. Puesto que primero se entregan los servicios de mayor prioridad y luego se integran los incrementos, los servicios de sistema más importantes reciben mayores pruebas. Esto

significa que los clientes tienen menos probabilidad de encontrar fallas de software en las partes más significativas del sistema.

Sin embargo, existen problemas con la entrega incremental:

1. La mayoría de los sistemas requieren de una serie de recursos que se utilizan para diferentes partes del sistema. Dado que los requerimientos no están definidos con detalle sino hasta que se implementa un incremento, resulta difícil identificar recursos comunes que necesiten todos los incrementos.
2. Asimismo, el desarrollo iterativo resulta complicado cuando se diseña un sistema de reemplazo. Los usuarios requieren de toda la funcionalidad del sistema antiguo, ya que es común que no deseen experimentar con un nuevo sistema incompleto. Por lo tanto, es difícil conseguir retroalimentación útil del cliente.
3. La esencia de los procesos iterativos es que la especificación se desarrolla en conjunto con el software. Sin embargo, esto se puede contradecir con el modelo de adquisiciones de muchas organizaciones, donde la especificación completa del sistema es parte del contrato de desarrollo del sistema. En el enfoque incremental, no hay especificación completa del sistema, sino hasta que se define el incremento final. Esto requiere una nueva forma de contrato que los grandes clientes, como las agencias gubernamentales, encontrarían difícil de adoptar.

Existen algunos tipos de sistema donde el desarrollo incremental y la entrega no son el mejor enfoque. Hay sistemas muy grandes donde el desarrollo incluye equipos que trabajan en diferentes ubicaciones, algunos sistemas embebidos donde el software depende del desarrollo de hardware y algunos sistemas críticos donde todos los requerimientos tienen que analizarse para comprobar las interacciones que comprometan la seguridad o protección del sistema. Estos sistemas, desde luego, enfrentan los mismos problemas de incertidumbre y requerimientos cambiantes. En consecuencia, para solucionar tales problemas y obtener algunos de los beneficios del desarrollo incremental, se utiliza un proceso donde un prototipo del sistema se elabore iterativamente y se utilice como plataforma, para experimentar con los requerimientos y el diseño del sistema. Con la experiencia obtenida del prototipo, pueden concertarse los requerimientos definitivos.

1.9. El Proceso Unificado de Rational

El Proceso Unificado de Rational (RUP, por las siglas de Rational Unified Process) es un es un buen ejemplo de un modelo de proceso híbrido. Incluye elementos de todos los modelos de proceso genéricos, ilustra la buena práctica en especificación y diseño, y apoya la creación de prototipos y entrega incremental. El RUP reconoce que los modelos de proceso convencionales presentan una sola visión del proceso. En contraste, el RUP por lo general se describe desde tres perspectivas:

1. Una perspectiva dinámica que muestra las fases del modelo a través del tiempo.
2. Una perspectiva estática que presenta las actividades del proceso que se establecen.
3. Una perspectiva práctica que sugiere buenas prácticas a usar durante el proceso.

El RUP es un modelo en fases que identifica cuatro fases discretas en el proceso de software. Sin embargo, a diferencia del modelo en cascada, donde las fases se igualan con actividades del proceso, las fases en el RUP están más estrechamente vinculadas con la empresa que con las preocupaciones técnicas. Las fases son:

1. Concepción. La meta de la fase de concepción es establecer un caso empresarial para el sistema. Deben identificarse todas las entidades externas (personas y sistemas) que

interactuarán con el sistema y definirán dichas interacciones. Luego se usa esta información para valorar el aporte del sistema hacia la empresa. Si este aporte es menor, entonces el proyecto puede cancelarse después de esta fase.

2. Elaboración. Las metas de la fase de elaboración consisten en desarrollar la comprensión del problema de dominio, establecer un marco conceptual arquitectónico para el sistema, diseñar el plan del proyecto e identificar los riesgos clave del proyecto. Al completar esta fase, debe tenerse un modelo de requerimientos para el sistema, que podría ser una serie de casos de uso del UML, una descripción arquitectónica y un plan de desarrollo para el software.
3. Construcción. La fase de construcción incluye diseño, programación y pruebas del sistema. Partes del sistema se desarrollan en paralelo y se integran durante esta fase. Al completar ésta, debe tenerse un sistema de software funcionando y la documentación relacionada y lista para entregarse al usuario.
4. Transición. La fase final del RUP se interesa por el cambio del sistema desde la comunidad de desarrollo hacia la comunidad de usuarios, y por ponerlo a funcionar en un ambiente real. Esto es algo ignorado en la mayoría de los modelos de proceso de software, aunque, en efecto, es una actividad costosa y en ocasiones problemática. En el complemento de esta fase se debe tener un sistema de software documentado que funcione correctamente en su entorno operacional.

La iteración con el RUP se apoya en dos formas. Cada fase puede presentarse en una forma iterativa, con los resultados desarrollados incrementalmente. Además, todo el conjunto de fases puede expresarse de manera incremental.

La visión estática del RUP se enfoca en las actividades que tienen lugar durante el proceso de desarrollo. Se les llama flujos de trabajo en la descripción RUP. En el proceso se identifican seis flujos de trabajo de proceso centrales y tres flujos de trabajo de apoyo centrales. El RUP se diseñó en conjunto con el UML, de manera que la descripción del flujo de trabajo se orienta sobre modelos UML asociados, como modelos de secuencia, modelos de objeto, etcétera. En la Tabla 1 se describen los flujos de trabajo.

La ventaja en la presentación de las visiones dinámica y estática radica en que las fases del proceso de desarrollo no están asociadas con flujos de trabajo específicos. En principio, al menos, todos los flujos de trabajo RUP pueden estar activos en la totalidad de las etapas del proceso. En las fases iniciales del proceso, es probable que se use mayor esfuerzo en los flujos de trabajo como modelado del negocio y requerimientos y, en fases posteriores, en las pruebas y el despliegue.

Flujo de Trabajo	Descripción
Modelado del negocio	Se modelan los procesos de negocios utilizando casos de uso de la empresa.
Requerimientos	Se identifican los actores que interactúan con el sistema y se desarrollan casos de uso para modelar los requerimientos del sistema.
Análisis y diseño	Se crea y documenta un modelo de diseño utilizando modelos arquitectónicos, de componentes, de objetos y de secuencias.
Implementación	Se implementan y estructuran los componentes del sistema en subsistemas de implementación. La generación automática

	de código a partir de modelos de diseño ayuda a acelerar este proceso.
Pruebas	Las pruebas son un proceso iterativo que se realiza en conjunto con la implementación. Las pruebas del sistema siguen al completar la implementación.
Despliegue	Se crea la liberación de un producto, se distribuye a los usuarios y se instala en su lugar de trabajo.
Administración de la configuración y del cambio	Este flujo de trabajo de apoyo gestiona los cambios al sistema.
Administración del proyecto	Este flujo de trabajo de apoyo gestiona el desarrollo del sistema.
Entorno	Este flujo de trabajo pone a disposición del equipo de desarrollo de software, las herramientas adecuadas de software.

Tabla 1. Flujos de trabajos en RUP

El enfoque práctico del RUP describe las buenas prácticas de ingeniería de software que se recomiendan para su uso en el desarrollo de sistemas. Las seis mejores prácticas fundamentales que se recomiendan son:

1. Desarrollo de software de manera iterativa. Incrementar el plan del sistema con base en las prioridades del cliente, y desarrollar oportunamente las características del sistema de mayor prioridad en el proceso de desarrollo.

2. Gestión de requerimientos. Documentar de manera explícita los requerimientos del cliente y seguir la huella de los cambios a dichos requerimientos. Analizar el efecto de los cambios sobre el sistema antes de aceptarlos.

3. Usar arquitecturas basadas en componentes. Estructurar la arquitectura del sistema en componentes.

4. Software modelado visualmente. Usar modelos UML gráficos para elaborar representaciones de software estáticas y dinámicas.

5. Verificar la calidad del software. Garantizar que el software cumpla con los estándares de calidad de la organización.

6. Controlar los cambios al software. Gestionar los cambios al software con un sistema de administración del cambio, así como con procedimientos y herramientas de administración de la configuración.

El RUP no es un proceso adecuado para todos los tipos de desarrollo, por ejemplo, para desarrollo de software embebido. Sin embargo, sí representa un enfoque que potencialmente combina los tres modelos de proceso genéricos descriptos anteriormente. Las innovaciones más importantes en el RUP son la separación de fases y flujos de trabajo, y el reconocimiento de que el despliegue del software en un entorno del usuario forma parte del proceso. Las fases son dinámicas y tienen metas. Los flujos de trabajo son estáticos y son actividades técnicas que no se asocian con una sola fase, sino que pueden usarse a lo largo del desarrollo para lograr las metas de cada fase.

Bibliografía

IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology, 1990.

Olivé, Antoni. Conceptual Modeling of Information Systems, Springer, 2007.

Pressman, Roger. Ingeniería de software, séptima edición, McGraw Hill, 2010.

Sommerville, Ian. Ingeniería de software, novena edición, Pearson, 2011.

Stair, Ralph; George Reynolds. Principles of Information Systems. A Managerial Approach, 9 edition, 2010.