Universidad Nacional del Litoral

FACULTAD DE INGENIERÍA
Y CIENCIAS HÍDRICAS

Ingeniería en Informática

Ingeniería de Software I

TEMA II – La ingeniería de software

Introducción

Es imposible operar en el mundo moderno sin software. Las infraestructuras nacionales y los servicios públicos se controlan mediante sistemas basados en computadoras y la mayoría de los productos eléctricos y electrónicos incluyen una computadora y un software de control. La fabricación y la distribución industrial está completamente computarizada como el sistema financiero, los entretenimientos, la industria musical, los juegos, el cine, la televisión, todos usan software de manera intensiva. Esto da una idea de lo esencial que es el software para el funcionamiento de la sociedad.

Los sistemas de software son abstractos e intangibles. No están restringidos por las propiedades de los materiales ni regidos por leyes físicas ni por procesos de fabricación. Esto simplifica al software porque no tiene límites naturales a su potencial, pero precisamente debido a la falta de restricciones físicas los sistemas de software se vuelven rápidamente complejos, difíciles de entender y costosos de cambiar.

Hay muchos tipos diferentes de sistemas de software por lo que no tiene sentido buscar notaciones, métodos o técnicas universales ya que estos distintos tipos requieren de diferentes enfoques y no todas requieren de las mismas técnicas.

Algunos sistemas de software no son complejos, sino que son las aplicaciones altamente intrascendentes que son especificadas, construidas, mantenidas y utilizadas por la misma persona, habitualmente el programador aficionado o el desarrollador profesional que trabaja en solitario. Esto no significa que todos estos sistemas sean toscos o poco elegantes, ni se pretende quitar mérito a sus creadores. Tales sistemas tienden a tener un propósito limitado y un ciclo de vida muy corto. Uno puede permitirse tirarlos a la basura y reemplazarlos por software completamente nuevo en lugar de intentar reutilizarlos, repararlos o extender su funcionalidad. El desarrollo de estas aplicaciones es generalmente más tedioso que difícil y aprender a diseñarlas no es algo que resulte de interés. En lugar de esto, interesan mucho más los desafíos que plantea el desarrollo del ***software de dimensión industrial***. Entre estos se encuentran aplicaciones que exhiben un conjunto muy rico de comportamientos, como ocurre por ejemplo en sistemas que dirigen o son dirigidos por eventos del mundo físico, aplicaciones que mantienen la integridad de cientos o miles de registros de información mientras permiten actualizaciones y consultas concurrentes y sistemas para la gestión y control de entidades del mundo real, tales como controladores de tráfico aéreo, ferroviario o bien aparatos de bioingeniería. Los sistemas de software de esta clase tienden a tener un ciclo de vida largo y a lo largo del tiempo muchos usuarios llegan a depender de su funcionamiento correcto. La característica distintiva del software de dimensión industrial es que resulta sumamente difícil, sino imposible, para el desarrollador individual comprender todas las sutilidades de su diseño; dicho de otra manera, la complejidad de tales sistemas excede la capacidad intelectual humana. La complejidad es una propiedad esencial de todos los sistemas de software de gran tamaño: se puede dominar pero no eliminar. Existen personas de genio con habilidades extraordinarias que pueden hacer el trabajo de varias otras personas, pero no son el común; el mundo está poblado de genios solamente en forma dispersa y no hay razón para creer que la comunidad de la ingeniería de software posee una proporción extraordinariamente grande de ellos. Aunque hay un toque de genialidad en todos nosotros, en el dominio del software de dimensión industrial no se puede confiar siempre en la inspiración divina, por lo tanto, hay que considerar vías de mayor disciplina para dominar la complejidad.

2.1 LA COMPLEJIDAD DEL SOFTWARE

La complejidad del software se deriva de cuatro problemas:

a. Complejidad del dominio del problema. Los problemas que intentan resolver con el software conllevan a menudo elementos de complejidad ineludible en los que se encuentran innumerables requisitos que compiten entre sí e incluso se contradicen. Existe un desacoplamiento entre los usuarios de un sistema y sus desarrolladores: los usuarios suelen encontrar grandes dificultades al intentar expresar con precisión sus necesidades en una forma que los desarrolladores puedan comprender. Se le suma a esto la perspectiva distinta del mismo problema que tienen estos actores. Otra complicación es que los requisitos de un sistema de software cambian frecuentemente durante su desarrollo especialmente porque la mera existencia de un proyecto de desarrollo de software altera las reglas del problema original.

b. Dificultad de gestionar el desarrollo. La tarea fundamental del equipo de desarrollo es la de dar vida a una ilusión de simplicidad de esa complejidad externa que tienen los usuarios. Esta cantidad de trabajo exige la utilización de equipos de desarrolladores y de forma ideal se utilizará un equipo lo más pequeño posible. Un mayor número de miembros implica una comunicación más compleja y por lo tanto una coordinación más difícil, particularmente si el equipo está disperso geográficamente.

c. Flexibilidad del software. Una empresa de construcción de edificios normalmente no gestiona su propia explotación forestal para cosechar árboles de los que obtendría la madera o bien tener una acería en la obra para hacer componentes a medida para el nuevo edificio. El software ofrece la flexibilidad máxima por lo que un desarrollador puede expresar casi cualquier clase de abstracción. Esta flexibilidad resulta ser una propiedad que seduce increíblemente y suele empujar al desarrollador a construir por sí mismo prácticamente todos los bloques fundamentales sobre los que se apoyan estas abstracciones de más alto nivel.

d. Caracterización de problemas discretos. En una aplicación de gran tamaño puede haber ciento o miles de variables así como más de un flujo posible de control. El conjunto de todas estas variables, sus valores actuales y la dirección de ejecución y pila actual de cada proceso del sistema constituyen el estado actual de la aplicación. En contraste, los sistema analógicos son sistemas continuos, mientras que en el software se maneja de manera determinista, o sea que se asume como un sistema discreto. Consecuentemente esta es la razón primaria para probar a fondo los sistemas para precisamente encontrar las interacciones que pueden existir entre eventos que no fueron tenidos en cuenta al discretizar.

Cuanto más complejo es un sistema, más abierto está al derrumbamiento total. Un constructor pensaría raramente en añadir un nuevo nivel de sótano para cocheras a un edificio ya construido de cien pisos; hacer tal cosa sería muy costoso e indudablemente sería un invitación al fracaso. Contrastando con esta situación, los usuarios de sistema de software casi nunca piensan dos veces a la hora de solicitar cambios equivalentes aduciendo que *simplemente es cuestión de programar*. El fracaso en dominar la complejidad del software lleva a proyectos retrasados, que exceden el presupuesto y que son deficientes respecto a los requerimientos fijados. Esta situación

es una parte de lo denominado ***Crisis del software***, pero en realidad es una enfermedad que ha existido siempre que debe considerarse crónica y casi normal.

La ingeniería de software busca apoyar el desarrollo de software profesional o de dimensión industrial en lugar de la programación individual. Incluye técnicas que apoyan la especificación, el diseño y la evolución del programa, ninguno de los cuales son normalmente relevantes para el desarrollo de software personal. Este conjunto de técnicas, métodos y herramientas posibilita precisamente "dominar la complejidad".

El término ingeniería de software no se refiere solamente al software propiamente dicho (programa de cómputo), sino también a toda la documentación asociada y los datos de configuración requeridos para hacer que estos programas operen de manera correcta. Esta es una de las principales diferencias entre el desarrollo de software profesional y el del aficionado.

2.2 TIPOS DE PRODUCTOS DE SOFTWARE

Los ingenieros de software desarrollan productos (software que se puede vender a un cliente) de alguno de los siguientes tipos:

1. Productos genéricos: sistemas independientes que se producen por una organización de desarrollo y se venden en el mercado abierto a cualquier cliente que desee comprarlo, por ejemplo software para PC de tipo bases de datos, procesadores de texto, paquetes de dibujo, etc. También aplicaciones de propósitos específicos tales como sistemas de contabilidad, sistemas de manejo de clínicas, etc. En éstos, la organización que desarrolla el software controla la especificación del mismo.
2. Productos personalizados (a medida): son sistemas que están destinados para un cliente en particular. Un contratista desarrolla especialmente para ese cliente. En éstos, la organización compradora controla la especificación por lo que los desarrolladores de software trabajan siguiendo esa especificación.

Sin embargo, esta distinción suele hacerse difusa en la actualidad ya que 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 ERP (Enterprise Resource Planning) como el sistema SAP son ejemplos de este caso. Un sistema grande y complejo se adapta a una compañía al incorporar la información acerca de las reglas y los procesos empresariales, los reportes, etc.

2.3 LA INGENIERÍA DE SOFTWARE

2.3.1 Evolución de la industria del software

El software es el factor decisivo a la hora de elegir entre varias soluciones informáticas disponibles para un problema dado, pero esto no ha sido siempre así. En los primeros años de la informática, el hardware tenía una importancia mucho mayor que en la actualidad. Su costo era comparativamente mucho más alto y su capacidad de almacenamiento y procesamiento, junto con su fiabilidad, era lo que limitaba las prestaciones de un determinado producto. El software

se consideraba como un simple añadido, a cuyo desarrollo se dedicaba poco esfuerzo y no se aplicaba ningún método sistemático. La programación era un arte de entrecasa y el desarrollo de software se realizaba sin ninguna planificación. La mayoría del software se desarrollaba y era utilizado por la misma persona u organización. La misma persona lo escribía, lo ejecutaba y, si fallaba, lo depuraba. Debido a que la movilidad en el trabajo era baja, los ejecutivos estaban seguros de que esa persona estaría allí cuando se encontrara algún error. Debido a este entorno personalizado del software, el diseño era un proceso implícito, realizado en la mente de alguien y la documentación normalmente no existía.

En este contexto, las empresas de informática se dedicaron a mejorar las prestaciones del hardware, reduciendo los costos y el consumo de los equipos, y aumentando la velocidad de cálculo y la capacidad de almacenamiento. Para ello, tuvieron que dedicar grandes esfuerzos a investigación y aplicaron métodos de ingeniería industrial al análisis, diseño, fabricación y control de calidad de los componentes de hardware. Como consecuencia de esto, el hardware se desarrolló rápidamente y la complejidad de los sistemas informáticos aumentó notablemente, necesitando de un software cada vez más complejo para su funcionamiento. Surgieron entonces las primeras casas de software, y el mercado se amplió considerablemente. Con ello aumentó la movilidad laboral, y con la marcha de un trabajador desaparecían las posibilidades de mantener o modificar los programas que éste había desarrollado. Al no utilizarse metodología alguna en el desarrollo del software, los programas contenían numerosos errores e inconsistencias, lo que obligaba a una depuración continua, incluso mucho después de haber sido entregados al cliente. Estas continuas modificaciones no hacían sino aumentar la inconsistencia de los programas, que se alejaban cada vez más de la corrección y se hacían prácticamente ininteligibles. Los costos se disparaban y frecuentemente era más rápido (y por tanto más barato) empezar de nuevo desde cero, desechando todo el trabajo anterior, que intentar adaptar un programa preexistente a un cambio de los requisitos. Sin embargo, los nuevos programas no estaban exentos de errores ni de futuras modificaciones, con lo que la situación volvía a ser la misma. Había comenzado la crisis del software.

Hoy en día, la distribución de costos en el desarrollo de sistemas informáticos ha cambiado drásticamente. El software, en lugar del hardware, es el elemento principal del costo. Esto ha hecho que las miradas de los ejecutivos de las empresas se centren en el software y a que se formulen las siguientes preguntas:

¿Por qué lleva tanto tiempo terminar los desarrollos?
¿Por qué es tan elevado el costo?
¿Por qué no es posible encontrar todos los errores antes de entregar el software al cliente?
¿Por qué resulta tan difícil constatar el progreso conforme se desarrolla el software?

Estas y otras muchas preguntas son una manifestación del carácter del software y de la forma en que se desarrolla y han llevado a la aparición y la adopción paulatina de la ingeniería del software.

2.3.2 Características del software

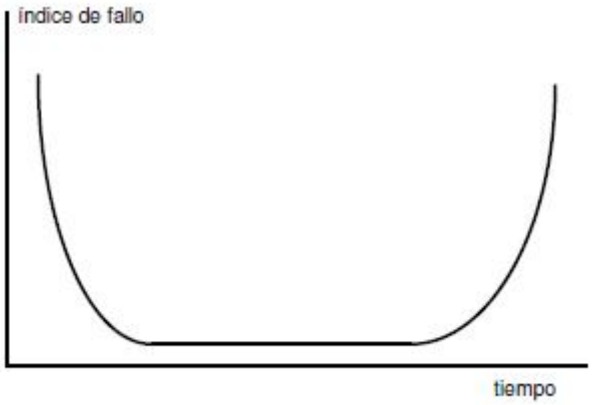
Algunas definiciones de software podrían ser:

- Instrucciones de computadora que cuando se ejecutan proporcionan la función y el comportamiento deseado,
- Estructuras de datos que facilitan a los programas manipular adecuadamente la información, y
- Documentos que describen la operación y el uso de los programas.

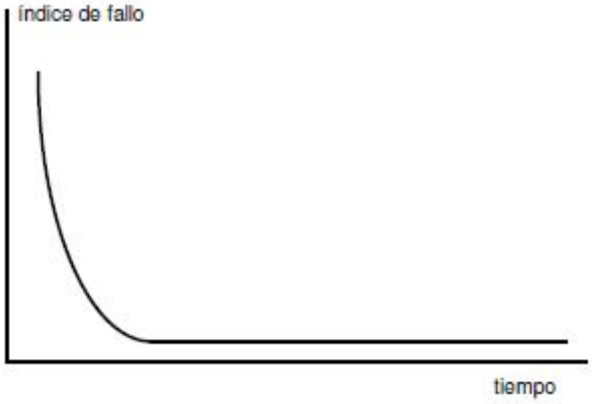
Por tanto, el software incluye no sólo los programas de computadora, sino también las estructuras de datos que manejan esos programas y toda la documentación que debe acompañar al proceso de desarrollo, mantenimiento y uso de dichos programas. Según esto, el software se diferencia de otros productos que los hombres puedan construir por su propia naturaleza lógica. En el desarrollo del hardware, el proceso creativo (análisis, diseño, construcción, prueba) se traduce finalmente en una forma material, en algo físico. Por el contrario, el software es inmaterial y por ello tiene unas características completamente distintas al hardware. Entre ellas podemos citar:

1. El software se desarrolla, no se fabrica en sentido estricto. Existen similitudes entre el proceso de desarrollo del software y el de otros productos industriales, como el hardware. En ambos casos existen fases de análisis, diseño y desarrollo o construcción, y la buena calidad del producto final se obtiene mediante un buen diseño. Sin embargo, en la fase de producción del software pueden producirse problemas que afecten a la calidad y que no existen, o son fácilmente evitables, en el caso del hardware. Por otro lado, en el caso de producción de hardware a gran escala, el costo del producto acaba dependiendo exclusivamente del costo de los materiales empleados y del propio proceso de producción, reduciéndose la repercusión en el costo de las fases de ingeniería previas. En el software, el desarrollo es una más de las labores de ingeniería, y la producción a gran o pequeña escala no influye en el impacto que tiene la ingeniería en el costo, al ser el producto inmaterial. Por otro lado, la replicación del producto (lo que sería equivalente a la producción del producto hardware) no presenta problemas técnicos, y no se requiere un control de calidad individualizado. Los costos del software se encuentran en la ingeniería (incluyendo en ésta el desarrollo), y no en la producción, y es ahí donde hay que incidir para reducir el costo final del producto.

2. El software no se estropea. Se pueden comparar las curvas de índices de fallos del hardware y el software en función del tiempo. En el caso del hardware (figura siguiente), se tiene la llamada curva de bañera, que indica que el hardware presenta relativamente muchos fallos al principio de su vida.

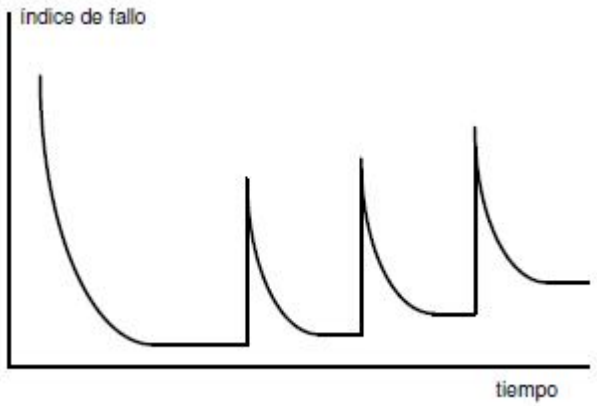


Estos fallos son debidos fundamentalmente a defectos de diseño o a la baja calidad inicial de la fase de producción. Una vez corregidos estos defectos, la tasa de fallos cae hasta un nivel estacionario y se mantiene así durante un cierto periodo de tiempo. Posteriormente, la tasa de fallos vuelve a incrementarse debido al deterioro de los componentes, que van siendo afectados por la suciedad, vibraciones y la influencia de muchos otros factores externos. Llegados a este punto, podemos sustituir los componentes defectuosos o todo el sistema por otros nuevos, y la tasa de fallos vuelve a situarse en el nivel estacionario. El software no es susceptible a los factores externos que hacen que el hardware se estropee. Por tanto la curva debería seguir la forma de la figura siguiente:



Inicialmente la tasa de fallos es alta, debido a la presencia de errores no detectados durante el desarrollo (los denominados errores latentes). Una vez corregidos estos errores, la tasa de fallos debería alcanzar el nivel estacionario y mantenerse ahí indefinidamente. Pero esto no es más que una simplificación del modelo real de fallos de un producto software. Durante su vida, el software sufre cambios, debidos al mantenimiento. El mantenimiento puede deberse a la corrección de errores latentes o a cambios en los requisitos iniciales del producto. Conforme se hacen cambios es bastante probable que se introduzcan nuevos errores, con lo que se producen picos en la curva de fallos. Estos errores pueden corregirse, pero el efecto de los sucesivos cambios hace que el producto se aleje cada vez más de las especificaciones iniciales de acuerdo a las cuales fue desarrollado, conteniendo cada vez más errores latentes. Además, con mucha frecuencia se solicita - y se emprende - un nuevo cambio antes de haber conseguido corregir todos los errores producidos por el cambio anterior. Por estas razones, el nivel

estacionario que se consigue después de un cambio es algo superior al que el que había antes de efectuarlo (figura siguiente), degradándose poco a poco el funcionamiento del sistema. Se puede decir entonces que el software no se estropea, pero se **deteriora**.



Además, cuando un componente software se deteriora, no se puede sustituir por otro, como en el caso del hardware ya que no existen piezas de repuesto. Cada fallo del software indica un fallo en el diseño o en el proceso mediante el cual se transformó el diseño en programación. La solución no es sustituir el componente defectuoso por otro (que sería idéntico y contendría los mismos errores) sino un nuevo diseño y desarrollo del producto. Por tanto, el mantenimiento del software tiene una complejidad mucho mayor que el mantenimiento del hardware.

3. La mayoría del software se construye a medida. El diseño de hardware se realiza, en gran medida, en base de componentes digitales existentes, cuyas características se comprueban en un catálogo y que han sido exhaustivamente probados por el fabricante y los usuarios anteriores. Estos componentes cumplen unas especificaciones claras y tienen unas interfaces definidas. El caso del software es totalmente distinto. No existen catálogos de componentes, y aunque determinados productos como sistemas operativos, editores, entornos de ventanas y bases de datos se venden en grandes ediciones, la mayoría del software se fabrica a medida, siendo la reutilización muy baja. Se puede comprar software ya desarrollado, pero sólo como unidades completas, no como componentes que pueden ser reensamblados para construir nuevos programas. Esto hace que el impacto de los costos de ingeniería sobre el producto final sea muy elevado, al dividirse entre un número de unidades producidas muy pequeño. Se ha escrito mucho sobre reutilización del software, y han sido muchos los intentos para conseguir aumentar el nivel de reutilización, normalmente con poco éxito. Uno de los ejemplos de reutilización más típicos, que ha venido usándose desde hace tiempo, son las bibliotecas. Durante los años sesenta se empezaron a desarrollar bibliotecas de subrutinas científicas, reutilizables en una amplia gama de aplicaciones científicas y de ingeniería. La mayor parte de los lenguajes modernos incluyen bibliotecas de este tipo, así como otras para facilitar la entrada/salida en los entornos de ventanas. Sin embargo esta aproximación no puede aplicarse fácilmente a otros problemas también de uso muy frecuente, como puede ser la búsqueda de un elemento en una estructura de datos, debido a la gran variación que existe en cuanto a la organización interna de estas estructuras y en la composición de los datos que contienen. Existen algoritmos para resolver estos problemas, pero no queda más

remedio que programarlos una y otra vez, adaptándolos a cada situación particular. Un nuevo intento de conseguir la reutilización se produjo con la utilización de técnicas de programación estructurada y modular. Sin embargo, se dedica por lo general poco esfuerzo al diseño de módulos lo suficientemente generales para ser reutilizables, y en todo caso, no se documentan ni se difunden todo lo que sería necesario para extender su uso, con lo que la tendencia habitual es diseñar y programar módulos muy semejantes una y otra vez. La programación estructurada permite diseñar programas con una estructura más clara, y que, por tanto, sean más fáciles de entender. Esta estructura interna más clara y estudiada, permite la reutilización de módulos dentro de los programas, o incluso dentro del proyecto que se está desarrollando, pero la reutilización de código en proyectos diferentes es muy baja. La tendencia actual para conseguir la reutilización es el uso de técnicas orientadas a objetos, que permiten la programación por especialización. Los objetos, que encapsulan tanto datos como los procedimientos que los manejan - los métodos -, haciendo los detalles de implementación invisibles e irrelevantes a quien los usa, disponen de interfaces claras, los errores cometidos en su desarrollo pueden ser depurados sin que esto afecte a la corrección de otras partes del código y pueden ser heredados y reescritos parcialmente, haciendo posible su reutilización aún en situaciones no contempladas en el diseño inicial. El software permite aplicaciones muy diversas, pero en todas ellas podemos encontrar algo en común: el objetivo es que el software desempeñe una determinada función, y además, debe hacerlo cumpliendo una serie de requisitos. Esos pueden ser muy variados: corrección, fiabilidad, respuesta en un tiempo determinado, facilidad de uso, bajo costo, etc., pero siempre existen y no podemos olvidarnos de ellos a la hora de desarrollar el software.

2.3.3 Problemas del software

Se mencionó la crisis del software. Sin embargo, por crisis se entiende normalmente un estado pasajero de inestabilidad, que tiene como resultado un cambio de estado del sistema o una vuelta al estado inicial, en caso de que se tomen las medidas para superarla. Teniendo en cuenta esto, el software, más que padecer una crisis podríamos decir que padece una enfermedad crónica. Los problemas que surgieron cuando se empezó a desarrollar software de una cierta complejidad siguen existiendo actualmente, sin que se haya avanzado mucho en los intentos de solucionarlos. Estos problemas son causados por las propias características del software y por los errores cometidos por quienes intervienen en su producción. Entre ellos podemos citar:

- <u>La planificación y la estimación de costos son muy imprecisas</u>
  A la hora de abordar un proyecto de una cierta complejidad, sea en el ámbito que sea, es frecuente que surjan imprevistos que no estaban recogidos en la planificación inicial, y como consecuencia de estos imprevistos se producirá una desviación en los costos del proyecto. Sin embargo, en el desarrollo de software lo más frecuente es que la planificación sea prácticamente inexistente, y que nunca se revise durante el desarrollo del proyecto. Sin una planificación detallada es totalmente imposible hacer una estimación de costos que tenga alguna posibilidad de cumplirse, y tampoco se pueden identificar las tareas conflictivas que pueden desviarnos de los costos previstos. Entre las causas de este problema se puede citar:

- No se recogen datos sobre el desarrollo de proyectos anteriores, con lo que no se acumula experiencia que pueda ser utilizada en la planificación de nuevos proyectos.
- Los administradores de proyectos no están especializados en la producción de software. Tradicionalmente, los responsables del desarrollo del software han sido ejecutivos de nivel medio y alto sin conocimientos de informática, siguiendo el principio de que *"un buen gestor (administrador, líder de proyecto, Project Manager) puede gestionar cualquier proyecto"*. Esto es cierto, pero no cabe duda de que también es necesario conocer las características específicas del software, aprender las técnicas que se aplican en su desarrollo y conocer una tecnología que evoluciona continuamente.

- La productividad es baja
  Los proyectos de software tienen, por lo general, una duración mucho mayor a la esperada. Como consecuencia de esto los costos se disparan y la productividad y los beneficios disminuyen. Uno de los factores que influyen en esto, es la falta de unos propósitos claros o realistas a la hora de comenzar el proyecto. La mayoría del software se desarrolla a partir de especificaciones ambiguas o incorrectas, y no existe apenas comunicación con el cliente hasta la entrega del producto. Debido a esto son muy frecuentes las modificaciones de las especificaciones sobre la marcha o los cambios de última hora, después de la entrega al cliente. No se realiza un estudio detallado del impacto de estos cambios y la complejidad interna de las aplicaciones crece hasta que se hacen virtualmente imposibles de mantener y cada nueva modificación, por pequeña que sea, es más costosa, y puede provocar el fallo de todo el sistema. Debido a la falta de documentación sobre cómo se ha desarrollado el producto o a que las sucesivas modificaciones - también indocumentadas - han desvirtuado totalmente el diseño inicial, el mantenimiento de software puede llegar a ser una tarea imposible de realizar, pudiendo llevar más tiempo el realizar una modificación sobre el programa ya escrito que analizarlo y desarrollarlo entero de nuevo.

- La calidad es mala
  Como consecuencia de que las especificaciones son ambiguas o incluso incorrectas, y de que no se realizan pruebas exhaustivas, el software contiene numerosos errores cuando se entrega al cliente. Estos errores producen un fuerte incremento de costos durante el mantenimiento del producto, cuando ya se esperaba que el proyecto estuviese acabado. Sólo recientemente se ha empezado a tener en cuenta la importancia de la prueba sistemática y completa, y han empezado a surgir conceptos como la fiabilidad y la garantía de calidad.

- El cliente queda insatisfecho
  Debido al poco tiempo e interés que se dedican al análisis de requisitos y a la especificación del proyecto, a la falta de comunicación durante el desarrollo y a la existencia de numerosos errores en el producto que se entrega, los clientes suelen quedar muy poco satisfechos de los resultados. Consecuencia de esto es que las aplicaciones tengan que ser diseñadas y desarrolladas de nuevo, que nunca lleguen a utilizarse o que se produzca con frecuencia un cambio de proveedor a la hora de abordar un nuevo proyecto.

2.3.4 Definición de ingeniería de software

El desarrollo de sistemas de software complejos no es una actividad trivial, que pueda abordarse sin una preparación previa. El considerar que un proyecto de desarrollo de software podía abordarse como cualquier otro ha llevado a una serie de problemas que limitan nuestra capacidad de aprovechar los recursos que el hardware pone a nuestra disposición. Los problemas tradicionales en el desarrollo de software no van a desaparecer de la noche a la mañana, pero identificarlos y conocer sus causas es el único método que nos puede llevar hacia su solución. No existe una fórmula mágica para solucionar estos problemas, pero combinando métodos aplicables a cada una de las fases del desarrollo de software, construyendo herramientas para automatizar estos métodos, utilizando técnicas para garantizar la calidad de los productos desarrollados y coordinando todas las personas involucradas en el desarrollo de un proyecto, podremos avanzar mucho en la solución de estos problemas. De ello se encarga la disciplina llamada Ingeniería del Software.

Ésta disciplina de la ingeniería 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 que se pone en operación.

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 no se considera muy efectivo (salvo en ciertas situaciones). El enfoque sistemático que usa la ingeniería de software se conoce 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 comunes a todos los procesos y son:

1. **Especificación** del software, donde clientes e ingenieros definen las funcionalidades del software que se producirá y las restricciones de su operación.
2. **Diseño e implementación** del software donde se diseña y programa cumpliendo con las especificaciones.
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 el mercado.

Diferentes tipos de sistemas necesitan distintos procesos de desarrollo. Por ejemplo, un software en tiempo real de una aeronave se debe especificar por completo antes de comenzar el desarrollo, mientras que en un sistema de comercio electrónico, la especificación y el programa por lo general se desarrollan en conjunto.

Comparación con otras ciencias afines

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 de software. El conocimiento de las ciencias de la computación es esencial para los ingenieros de software (como la física para un ingeniero

electrónico). La ingeniería en 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 en sistemas se preocupa por el desarrollo del hardware, el diseño de políticas y procesos, la implantación del sistema así como por la ingeniería de software. Los ingenieros en sistemas intervienen en la especificación del sistema definiendo su arquitectura global y están menos preocupados por la ingeniería de los componentes de ese sistema.

La ingeniería de software es un enfoque sistemático para la producción de software considerando elementos prácticos tales como costos y planificación. Éste enfoque sistemático varía mucho dependiendo de la organización que desarrolla el software y el tipo de software por lo que no existen métodos y técnicas universales de ingeniería de software que sean adecuados para todos los sistemas y organizaciones. Uno de los factores más significativos en la determinación de qué método o técnica aplicar es el tipo de aplicación. Estos tipos pueden ser:

1. **Aplicaciones independientes**. Se trata de sistemas de aplicación que corren en una PC local e incluyen toda la funcionalidad necesaria sin necesidad de conectarse a una red.
2. **Aplicaciones interactivas basadas en transacción**. Son las aplicaciones que se ejecutan remotamente y los usuarios acceden desde su propia PC o terminal. Se incluyen las aplicaciones WEB como comercio electrónico o sistemas empresariales.
3. **Sistemas de control embebido**. Sistemas de control de software que regulan y gestionan dispositivos de hardware. Son los más numerosos. Por ejemplo los software de celulares, los frenos ABS, una video, etc.
4. **Sistema de procesamiento por lotes**. Batch. Procesan gran cantidad de entradas individuales para crear salidas correspondientes. Son por ejemplo los sistemas de facturación periódica como los de facturas telefónicas, impuestos, sueldos, etc.
5. **Sistemas de entretenimiento**. Sistemas de uso personal. Juegos.
6. **Sistemas para modelado y simulación**. Sistemas que desarrollan científicos o ingenieros para modelar procesos o situaciones físicas que incluyen muchos objetos separados interactuantes. Son computacionalmente intensivos y requieren muchos recursos de hardware.
7. **Sistemas de adquisición de datos**. Son sistemas que desde su entorno recopilan datos mediante sensores y los envían para su procesamiento a otro sistema. Interactúan con sensores y se instalan en ambientes hostiles.

Si bien cada situación tal vez amerite una metodología distinta, existen fundamentos de la ingeniería de software que se aplican a todos los sistemas de software:

- 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 se producirá y el tiempo en que estará terminado. Se usan diferentes procesos para distintos tipos de software.
- La confiabilidad y el desempeño son importantes para todos los tipos de sistemas. El software tiene que comportarse como se espera, sin fallas y cando se requiera estar disponible. Debe ser seguro en su operación y contra ataques externos. Debe desempeñarse de manera eficiente y no desperdiciar recursos.
- Es importante comprender y gestionar la especificación y los requerimientos del

software (lo que 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.

- Tiene que usar de manera tan efectiva como sea posible los recursos existentes. Donde sea adecuado se debe reutilizar el software ya desarrollado en vez de diseñar uno nuevo.

La ingeniería del software abarca un conjunto de tres elementos clave: métodos, herramientas y procedimientos, que faciliten al project manager (PM, líder de proyecto) el control del proceso de desarrollo y suministren a los implementadores bases para construir de forma productiva software de alta calidad.

Los métodos indican cómo construir técnicamente el software, y abarcan una amplia serie de tareas que incluyen la planificación y estimación de proyectos, el análisis de requisitos, el diseño de estructuras de datos, programas y procedimientos, la codificación, las pruebas y el mantenimiento. Los métodos introducen frecuentemente una notación específica para la tarea en cuestión y una serie de criterios de calidad.

Las herramientas proporcionan un soporte automático o semiautomático para utilizar los métodos. Existen herramientas automatizadas para cada una de las fases vistas anteriormente, y sistemas que integran las herramientas de cada fase de forma que sirven para todo el proceso de desarrollo. Estas herramientas se denominan CASE (Computer Assisted Software Engineering).

Los procedimientos definen la secuencia en que se aplican los métodos, los documentos que se requieren, los controles que permiten asegurar la calidad y las directrices que permiten a los PM evaluar los progresos.

2.4 MODELOS DE PROCESO DE SOFTWARE (ciclo de vida)

No existe un proceso ***ideal***. La mayoría de las organizaciones han diseñado sus propios procesos de desarrollo de software. Los procesos han evolucionado para beneficiarse de las capacidades de los integrantes de la organización y de las características específicas de los sistemas que se están desarrollando. Para algunos 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 primeros son aquellos donde las actividades del proceso se planean por anticipado y el avance se mide contra dicho plan. En los ágiles la planeación es incremental y es más fácil de modificar el proceso para reflejar los requerimientos cambiantes del cliente. Cada enfoque es adecuado para diferentes tipos de software por lo que se requiere encontrar un equilibrio entre procesos dirigidos por un plan y procesos ágiles.

Por ciclo de vida, se entiende la sucesión de etapas por las que pasa el software desde que un nuevo proyecto es concebido hasta que se deja de usar. Cada una de estas etapas lleva asociada una serie de tareas o actividades que deben realizarse, y una serie de documentos (en sentido amplio: software) que serán la salida de cada una de estas fases y servirán de entrada en la

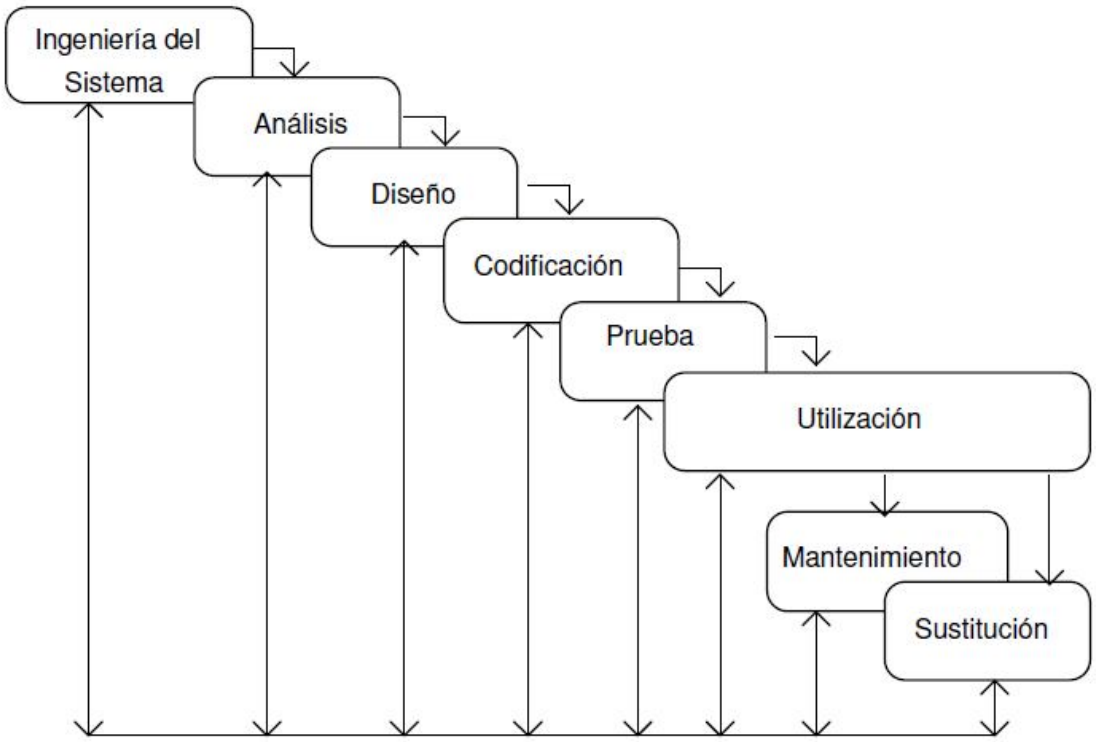
fase siguiente. Existen diversos modelos de proceso de software o de ciclo de vida, es decir, diversas formas de ver el proceso de desarrollo de software, y cada uno de ellos va asociado a un paradigma de la ingeniería del software, es decir, a una serie de métodos, herramientas y procedimientos que debemos usar a lo largo de un proyecto. Los modelos que se verán son:

- Modelo en cascada
- Desarrollo incremental o evolutivo
- Ingeniería de software orientada a la reutilización.

2.4.1 El modelo en cascada (waterfall) o ciclo de vida clásico

Este paradigma es el más antiguo de los empleados en la ingeniería de software y se desarrolló a partir del ciclo convencional de una ingeniería. No hay que olvidar que la ingeniería de software surgió como copia de otras ingenierías, especialmente de la del hardware, para dar solución a los problemas más comunes que aparecían al desarrollar sistemas de software complejos.

Es un ciclo de vida en sentido amplio, que incluye no sólo las etapas de ingeniería sino toda la vida del producto: las pruebas, el uso (la vida útil del software) y el mantenimiento, hasta que llega el momento de sustituirlo.



El ciclo de vida en cascada exige un enfoque sistemático y secuencial del desarrollo de software, que comienza en el nivel de la ingeniería de sistemas y avanza a través de fases secuenciales sucesivas. Estas fases son las siguientes:

Etapa 1: Ingeniería y análisis del sistema

El software es siempre parte de un sistema mayor, por lo que siempre va a interrelacionarse con otros elementos, ya sea hardware, otro sistema de software o personas. Por esto, el primer paso del ciclo de vida de un proyecto consiste en un análisis de las características y el comportamiento del sistema del cual el software va a formar parte. En el caso de que se desee construir un sistema nuevo, por ejemplo un sistema de control, se debe analizar cuáles son los requisitos y la función del sistema, y luego asignarle un subconjunto de estos requisitos al software. En el caso de un sistema ya existente (por ejemplo si se quiere informatizar una empresa) se debe analizar el funcionamiento de la misma, - las operaciones que se llevan a cabo en ella -, y asignarle al software aquellas funciones que se van a automatizar. La ingeniería del sistema comprende, por tanto, los requisitos globales a nivel del sistema, así como una cierta cantidad de análisis y de diseño a nivel superior, es decir sin entrar en mucho detalle.

Etapa 2: Análisis de requisitos del software

El análisis de requisitos debe ser más detallado para aquellos componentes del sistema que se van a implementar mediante software. El ingeniero del software debe comprender cuáles son los datos que se van a manejar, cuál va a ser la función que tiene que cumplir el software, cuáles son las interfaces requeridas y cuál es el rendimiento que se espera lograr. Los requisitos, tanto del sistema como del software deben documentarse y revisarse con el cliente.

Etapa 3: Diseño

El diseño se aplica a cuatro características distintas del software:

- la estructura de los datos
- la arquitectura de las aplicaciones
- la estructura interna de los programas
- las interfaces

El diseño es el proceso que traduce los requisitos en una representación del software de forma que pueda conocerse la arquitectura, funcionalidad e incluso la calidad del mismo antes de comenzar la codificación. Al igual que el análisis, el diseño debe documentarse y forma parte de la configuración del software (el control de configuraciones es lo que nos permite realizar cambios en el software de forma controlada y no traumática para el cliente).

Etapa 4: Codificación

La codificación consiste en la traducción del diseño a un formato que sea legible para la máquina. Si el diseño es lo suficientemente detallado, la codificación es relativamente sencilla, y puede hacerse - al menos en parte - de forma automática, usando generadores de código. Se puede observar que estas primeras fases del ciclo de vida consisten básicamente en una traducción: del análisis del sistema, los requisitos, la función y la estructura de este se traducen a un documento de análisis del sistema que está formado en parte por diagramas y en parte por descripciones en lenguaje natural. En el análisis de requisitos se profundiza en el estudio del componente software del sistema y esto se traduce a un documento, también formado por diagramas y descripciones en lenguaje natural. En el diseño, los requisitos del software se traducen a una serie de diagramas que representan la estructura del sistema software, de sus datos, de sus programas y de sus interfaces. Por

último, en la codificación se traducen estos diagramas de diseño a un lenguaje fuente, que luego se traduce - se compila - para obtener un programa ejecutable.

Etapa 5: Prueba

Una vez que se tiene el programa ejecutable, comienza la fase de pruebas. El objetivo es comprobar que no se hayan producido errores en alguna de las fases de traducción anteriores, especialmente en la codificación. Para ello deben probarse todas las sentencias, no sólo los casos normales y todos los módulos que forman parte del sistema.

Etapa 6: Utilización

Una vez superada la fase de pruebas, el software se entrega al cliente y comienza la vida útil del mismo. La fase de utilización se solapa con las posteriores - el mantenimiento, evolución y la sustitución - y dura hasta que el software, ya reemplazado por otro, deje de utilizarse.

Etapa 7: Mantenimiento

El software sufrirá cambios a lo largo de su vida útil. Estos cambios pueden ser debidos a tres causas:

- Que, durante la utilización, el cliente detecte errores en el software (los errores latentes).
- Que se produzcan cambios en alguno de los componentes del sistema informático: por ejemplo cambios en la máquina, en el sistema operativo o en los periféricos.
- Que el cliente requiera modificaciones funcionales (normalmente ampliaciones) no contempladas en el proyecto.

En cualquier caso, el mantenimiento supone volver atrás en el ciclo de vida, a las etapas de codificación, diseño o análisis dependiendo de la magnitud del cambio.
El modelo en cascada, a pesar de ser lineal, contiene flujos que permiten la vuelta atrás. Así, desde el mantenimiento se vuelve al análisis, el diseño o la codificación, y también desde cualquier fase se puede volver a la anterior si se detectan fallos. Estas vueltas atrás no son controladas, ni quedan explícitas en el modelo, y este es uno de los problemas que presenta este paradigma.

Etapa 8: Sustitución

La vida del software no es ilimitada y cualquier aplicación, por buena que sea, acaba por ser sustituida por otra más amplia, más rápida o más bonita y fácil de usar. La sustitución de un software que está funcionando por otro que acaba de ser desarrollado es una tarea que hay que planificar cuidadosamente y que hay que llevar a cabo de forma organizada. Es conveniente realizarla por fases, si esto es posible, no sustituyendo todas las aplicaciones de golpe, puesto que la sustitución conlleva normalmente un aumento de trabajo para los usuarios, que tienen que acostumbrarse a las nuevas aplicaciones, y también para los implementadores, que tienen que corregir los errores que aparecen. Es necesario hacer una migración de la información que maneja el sistema viejo a la estructura y el formato requeridos por el nuevo. Además, es conveniente mantener los dos sistemas funcionando en paralelo durante algún tiempo para comprobar que el sistema nuevo funcione correctamente y para asegurar el funcionamiento normal de la organización aún en el caso de que el sistema nuevo falle y tenga que volverse a alguna de las fases de desarrollo. La

sustitución implica el desarrollo de programas para la interconexión de ambos sistemas, el viejo y el nuevo, y para la migración y eventualmente la replicación de los datos entre ambos, evitando la duplicación del trabajo de las personas encargadas del proceso de datos durante el tiempo en que ambos sistemas funcionen en paralelo.

El ciclo de vida en cascada es el paradigma más antiguo, más conocido y más ampliamente usado en la ingeniería de software. No obstante, ha sufrido diversas críticas, debido a los problemas que se plantean al intentar aplicarlo a determinadas situaciones. Entre estos problemas están:

- En la realidad los proyectos no siguen un ciclo de vida estrictamente secuencial como propone el modelo. Siempre hay iteraciones. El ejemplo más típico es la fase de mantenimiento, que implica siempre volver a alguna de las fases anteriores, pero también es muy frecuente que en una fase, por ejemplo el diseño, se detecten errores que obliguen a volver a la fase anterior, el análisis.
- Es difícil que se puedan establecer inicialmente todos los requisitos del sistema. Normalmente los clientes no tienen conocimiento de la importancia de la fase de análisis o bien no han pensado en todo detalle qué es lo que quieren que haga el software. Los requisitos se van aclarando y refinando a lo largo de todo el proyecto, según se plantean dudas concretas en el diseño o la codificación pero el ciclo de vida clásico requiere la definición inicial de todos los requisitos y no es fácil acomodar en él las incertidumbres que suelen existir al comienzo de todos los proyectos.
- Hasta que se llega a la fase final del desarrollo: la codificación, no se dispone de una versión operativa de las aplicaciones. Como la mayor parte de los errores se detectan cuando el cliente puede probar los programas no se detectan hasta el final del proyecto, cuando son más costosos de corregir y más prisa (y más presiones) hay por que el sistema se ponga definitivamente en marcha.

Todos estos problemas son reales, pero de todas formas es mucho mejor desarrollar software siguiendo el modelo de ciclo de vida en cascada que hacerlo sin ningún tipo de guías. Además, este modelo describe una serie de pasos genéricos que son aplicables a cualquier otro paradigma, refiriéndose la mayor parte de las críticas que recibe a su carácter secuencial.

2.4.2 El modelo de desarrollo incremental o evolutivo

Dos de las críticas que se hacían al modelo de ciclo de vida en cascada eran que es difícil tener claros todos los requisitos del sistema al inicio del proyecto, y que no se dispone de una versión operativa de la o las aplicaciones hasta las fases finales del desarrollo, lo que dificulta la detección de errores y deja también para el final el descubrimiento de los requisitos inadvertidos en las fases de análisis. Para paliar estas deficiencias se ha propuesto un modelo de ciclo de vida evolutivo.

El desarrollo evolutivo se basa en la idea de desarrollar una implementación inicial exponiéndola a los comentarios del usuario y refinándola a través de las diferentes versiones hasta que se desarrolla un sistema adecuado. Las actividades de especificación, desarrollo y validación se

entrelazan en vez de separarse con una rápida retroalimentación entre éstas. Existen dos tipos de desarrollo evolutivo:

1. **Desarrollo exploratorio:** donde el objetivo del proceso es trabajar con el cliente para explorar sus requerimientos y entregar un sistema final. El desarrollo empieza con las partes del sistema que se comprenden mejor. El sistema evoluciona agregando nuevos atributos propuestos por el cliente.
2. **Prototipos desechables:** donde el objetivo del proceso de desarrollo evolutivo es comprender los requerimientos del cliente y entonces desarrollar una definición mejorada de los requerimientos del sistema. El prototipo se centra en experimentar con los requerimientos del cliente que no se comprenden del todo.

En la producción de sistemas, un enfoque evolutivo para el desarrollo de software suele ser más efectivo que el enfoque en cascada ya que satisface las necesidades inmediatas de los clientes. La ventaja de un proceso del software que se basa en un enfoque evolutivo es que la especificación se puede desarrollar en forma creciente. Tan pronto como los usuarios desarrollen un mejor entendimiento de su problema, éste se puede reflejar en el sistema software. El desarrollo de software incremental es la parte fundamental de los enfoques ágiles y es mucho mejor que el en cascada para la mayoría de los sistemas empresariales de comercio electrónico y personales. El desarrollo incremental refleja la forma en que se resuelven los problemas; rara vez se trabaja por adelantado una solución completa del problema sino que más bien se avanza en una serie de pasos hacia una solución y se retrocede cuando se detecta que se cometieron errores. 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 más urgente. De esta manera 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 se cambia y posiblemente se defina una nueva función para incrementos posteriores.

Para sistemas grandes, se recomienda un proceso mixto que incorpore las mejores características del modelo en cascada y del desarrollo evolutivo. Esto puede implicar desarrollar un prototipo desechable utilizando un enfoque evolutivo para resolver incertidumbres en la especifi-

cación del sistema. Puede entonces reimplementarse utilizando un enfoque más estructurado. Las partes del sistema bien comprendidas se pueden especificar y desarrollar utilizando un proceso basado en el modelo de cascada. Las otras partes, como la interfaz de usuario que son difíciles de especificar por adelantado, se deben desarrollar siempre utilizando un enfoque de desarrollo exploratorio.

En general, cualquier aplicación que presente mucha interacción con el usuario, o que necesite algoritmos que puedan construirse de manera evolutiva, yendo de lo más general a lo más específico es un buen candidato para este tipo de modelo. No obstante, hay que tener en cuenta la complejidad: si la aplicación necesita que se desarrolle una gran cantidad de código para poder tener un prototipo para mostrar al usuario, las ventajas de la construcción de prototipos se verán superadas por el esfuerzo de desarrollar un prototipo que al final habrá que desechar o modificar mucho. También hay que tener en cuenta la predisposición del cliente para probar un prototipo y sugerir modificaciones de los requisitos. Puede ser que el cliente "no tenga tiempo para andar jugando" o "no vea las ventajas de este método de desarrollo".

También es conveniente construir prototipos para probar la eficiencia de los algoritmos que se van a implementar, o para comprobar el rendimiento de un determinado componente del sistema, por ejemplo, una base de datos o el soporte hardware, en condiciones similares a las que existirán durante la utilización del sistema. Es bastante frecuente que el producto de ingeniería desarrollado presente un buen rendimiento durante la fase de pruebas realizada por los ingenieros antes de entregarlo al cliente (pruebas que se realizarán normalmente con unos pocos registros en la base de datos o pocos clientes conectados al sistema), pero que sea muy ineficiente, o incluso inviable, a la hora de almacenar o procesar el volumen real de información que producirá su trabajo en el ambiente de producción real. En estos casos, la construcción de un prototipo de parte del sistema y la realización de pruebas de rendimiento, sirven para decidir, antes de empezar la fase de diseño, cuál es el modelo más adecuado de entre la gama disponible para el soporte hardware o cómo deben hacerse los accesos a la base de datos para obtener buenas respuestas en tiempo cuando la aplicación esté ya en funcionamiento.

En otros casos, el prototipo servirá para modelar y poder mostrar al cliente cómo va a realizarse la entrada y salida de datos en la aplicación, de forma que éste pueda hacerse una idea de como va a ser el sistema final, pudiendo entonces detectar deficiencias o errores en la especificación aunque el modelo no sea más que una cáscara vacía.

El prototipado es una técnica que sirve fundamentalmente para la fase de análisis de requisitos, pero lleva consigo la obtención de una serie de subproductos que son útiles a lo largo del desarrollo del proyecto ya que parte del trabajo realizado durante la fase de diseño rápido (especialmente la definición de pantallas e informes) puede ser utilizada durante el diseño del producto final. Además, tras realizar varias vueltas en el ciclo de construcción de prototipos, el diseño del mismo se parece cada vez más al que tendrá el producto final. Además durante la fase de construcción de prototipos será necesario codificar algunos componentes del software que también podrán ser reutilizados en la codificación del producto final, aunque deban ser optimizados en cuanto a corrección o velocidad de procesamiento.

No obstante, hay que tener en cuenta que el prototipo **no es el sistema final**, puesto que normalmente apenas es utilizable. Será demasiado lento, demasiado grande, inadecuado para el

volumen de datos necesario, contendrá errores (debido al diseño rápido), será demasiado general (sin considerar casos particulares, que debe tener en cuenta el sistema final) o estará codificado en un lenguaje o para una máquina inadecuadas, o a partir de componentes software previamente existentes. No hay que preocuparse de haber desperdiciado tiempo o esfuerzos construyendo prototipos que luego habrán de ser desechados si con ello se ha conseguido tener más clara la especificación del proyecto, puesto que el tiempo perdido se ahorrará en las fases siguientes, que podrán realizarse con menos esfuerzo y en las que se cometerán menos errores que nos obliguen a volver atrás en el ciclo de vida.

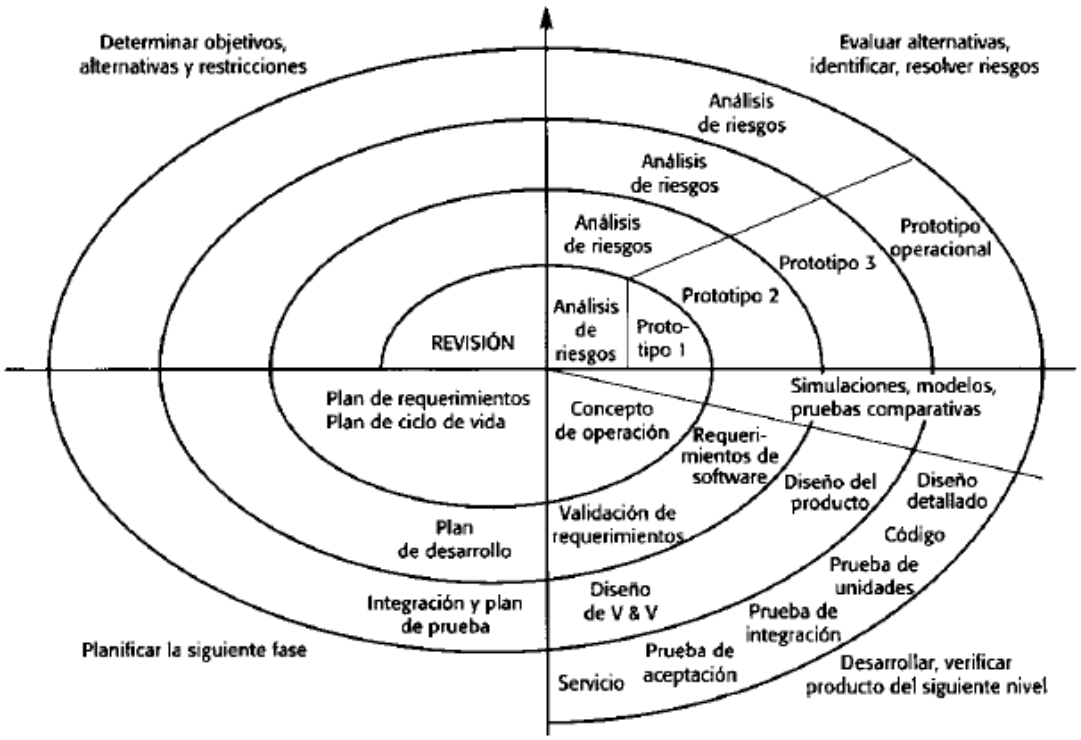
Hay que tener en cuenta que un análisis de requisitos incorrecto o incompleto, cuyos errores y deficiencias se detecten a la hora de las pruebas o tras entregar el software al cliente, obligará a repetir de nuevo las fases de análisis, diseño y codificación, que tal vez se habían realizado cuidadosamente, pensando que se estaba desarrollando el producto final. Al tener que repetir estas fases, sí que estaremos desechando una gran cantidad de trabajo, normalmente muy superior al esfuerzo de construir un prototipo basándose en un diseño rápido, en la reutilización de trozos de software preexistentes y en herramientas de generación de código para informes y manejo de ventanas.

Uno de los problemas que suelen aparecer siguiendo el paradigma de construcción de prototipos, es que con demasiada frecuencia el prototipo pasa a ser parte del sistema final, bien sea por presiones del cliente, que quiere tener el sistema funcionando lo antes posible o bien porque los técnicos se han acostumbrado a la máquina, el sistema operativo o el lenguaje con el que se desarrolló el prototipo. Se olvida aquí que el prototipo ha sido construido de forma acelerada, sin tener en cuenta consideraciones de eficiencia, calidad del software o facilidad de mantenimiento, o que las elecciones de lenguaje, sistema operativo o máquina para desarrollarlo se han hecho basándose en criterios como el mejor conocimiento de esas herramientas por parte los técnicos que en que sean adecuadas para el producto final.

2.4.3 El modelo en espiral de Boehm

El modelo en espiral combina las principales ventajas del modelo de ciclo de vida en cascada y del modelo de construcción de prototipos. Proporciona un modelo evolutivo para el desarrollo de sistemas de software complejos, mucho más realista que el ciclo de vida clásico, y permite la utilización de prototipos en cualquier etapa de la evolución del proyecto. Es un modelo de proceso dirigido por el riesgo. Aquí el proceso de software se presenta como una espiral y no como una secuencia de actividades con cierto retroceso de una actividad a otra. Cada ciclo en la espiral representa una fase del proceso de software, de esta manera, el ciclo más interno puede relacionarse con la factibilidad del sistema, el siguiente con la definición de requerimientos, el siguiente con el diseño del sistema, etc.

La reducción de riesgos se realiza mediante los prototipos, permitiendo finalizar el proyecto antes de haberse embarcado en el desarrollo del producto final (si el riesgo es demasiado grande).



El modelo en espiral define cuatro tipos de actividades, y representa cada uno de ellos en un cuadrante:

Establecimiento de objetivos y planificación

Consiste en determinar los objetivos del proyecto, las posibles alternativas y las restricciones. Esta fase equivale a la de recolección de requisitos del ciclo de vida clásico e incluye además la planificación de las actividades a realizar en cada iteración.

Análisis de riesgo

En cada uno de los riesgos identificados del proyecto, se realiza un análisis minucioso. Se dan acciones para reducir el riesgo. Por ejemplo, si existe un riesgo que los requerimientos sean inadecuados, puede desarrollarse un sistema prototipo.

Desarrollo y validación

Después de una evaluación del riesgo, se elige un modelo de desarrollo para el sistema. Por ejemplo, la creación de prototipos desechables sería el mejor enfoque de desarrollo si predominan los riesgos en la interfaz del usuario. Si el principal riesgo identificado es la integración de subsistemas, el modelo en cascada sería el mejor modelo de desarrollo a utilizar.

Planeación

El proyecto se revisa y se toma una decisión sobre si hay que continuar con otro ciclo de la espiral. Si se opta por continuar, se trazan los planes para la siguiente fase del proyecto.

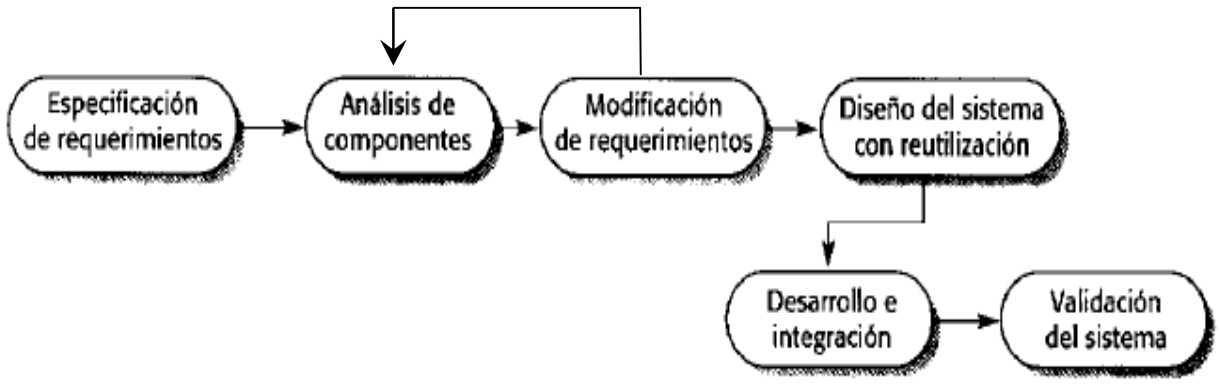
En la primera iteración se definen los requisitos del sistema y se realiza la planificación inicial del mismo. A continuación se analizan los riesgos del proyecto, basándonos en los requisitos iniciales y se procede a construir un prototipo del sistema. Entonces el cliente procede a evaluar el prototipo y con sus comentarios, se procede a refinar los requisitos y a reajustar la planificación inicial, volviendo a empezar el ciclo. En cada una de las iteraciones se realiza el análisis de riesgos, teniendo en cuenta los requisitos y la reacción del cliente ante el último prototipo. Si los riesgos son demasiado grandes se terminará el proyecto, aunque lo normal es que se siga avanzando a lo largo de la espiral.

Con cada iteración, se construyen sucesivas versiones del software, cada vez más completas, y aumenta la duración de las operaciones del cuadrante de ***Desarrollo y validación***, obteniéndose al final el sistema de ingeniería completo.

La diferencia principal con el modelo de construcción de prototipos, es que en éste los prototipos se usan para perfilar y definir los requisitos. Al final, el prototipo se desecha y comienza el desarrollo del software siguiendo el ciclo clásico. En el modelo en espiral, en cambio, los prototipos son sucesivas versiones del producto, cada vez más detalladas (el último es el producto en sí) y constituyen el esqueleto del producto de ingeniería razón por la cual deben construirse siguiendo estándares de calidad.

2.4.4 Ingeniería de software orientada a la reutilización

En la mayoría de 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 similares a los requeridos. Los buscan, modifican e incorporan en sus sistemas. Esta utilización ocurre independientemente del proceso de desarrollo que se emplee. En ocasiones tales componentes son sistemas por derecho propio (sistemas comerciales, ERP, procesadores de texto, hojas de cálculo) que pueden mejorar la funcionalidad específica.



Las etapas de especificación de requerimientos y la de validación son similares a otros procesos de software, las etapas intermedias son diferentes. Estas etapas son:

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 usa proporcionan sólo parte de la funcionalidad requerida.

Modificación de requerimientos

En 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.

Diseño de sistema con reutilización

En esta etapa se diseña el marco conceptual del sistema o se reutiliza el 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.

Desarrollo e integración

Se diseña el software que no puede procurarse de manera externa y se integran los componentes y sistemas comerciales para crear el nuevo sistema.

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 el marco de componentes como J2EE o .NET.
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 consecuentemente disminuir costos y riesgos. Por lo general conduce a entregas más rápidas de software, pero puede conducir hacia un sistema que no cubra las necesidades reales de los usuarios. Además se pierde el control sobre la evolución del sistema conforme las nuevas versiones de los componentes reutilizables no estén bajo el control de la organización que lo usa.