INGENIERÍA DE SOFTWARE
PROGRAMADOR UNIVERSITARIO

INGENIERÍA DE SOFTWARE II
LICENCIATURA EN INFORMÁTICA

AÑO 2014

LIC. MARÍA ISABEL MENTZ

FACULTAD DE CIENCIAS EXACTAS Y TECNOLOGÍA
UNIVERSIDAD NACIONAL DE TUCUMAN

2. Técnicas de Prueba del Software

En oposición a la tarea constructiva que viene desarrollando el Ingeniero de Software hasta este punto, esa nueva etapa de prueba del software tiene un **sentido destructivo**, en cuanto se crean un conjunto de casos de prueba que intentan "demoler" el software que ha sido construido.

Glenn Myers, en su libro sobre prueba de software establece los siguientes **objetivos para la prueba:**

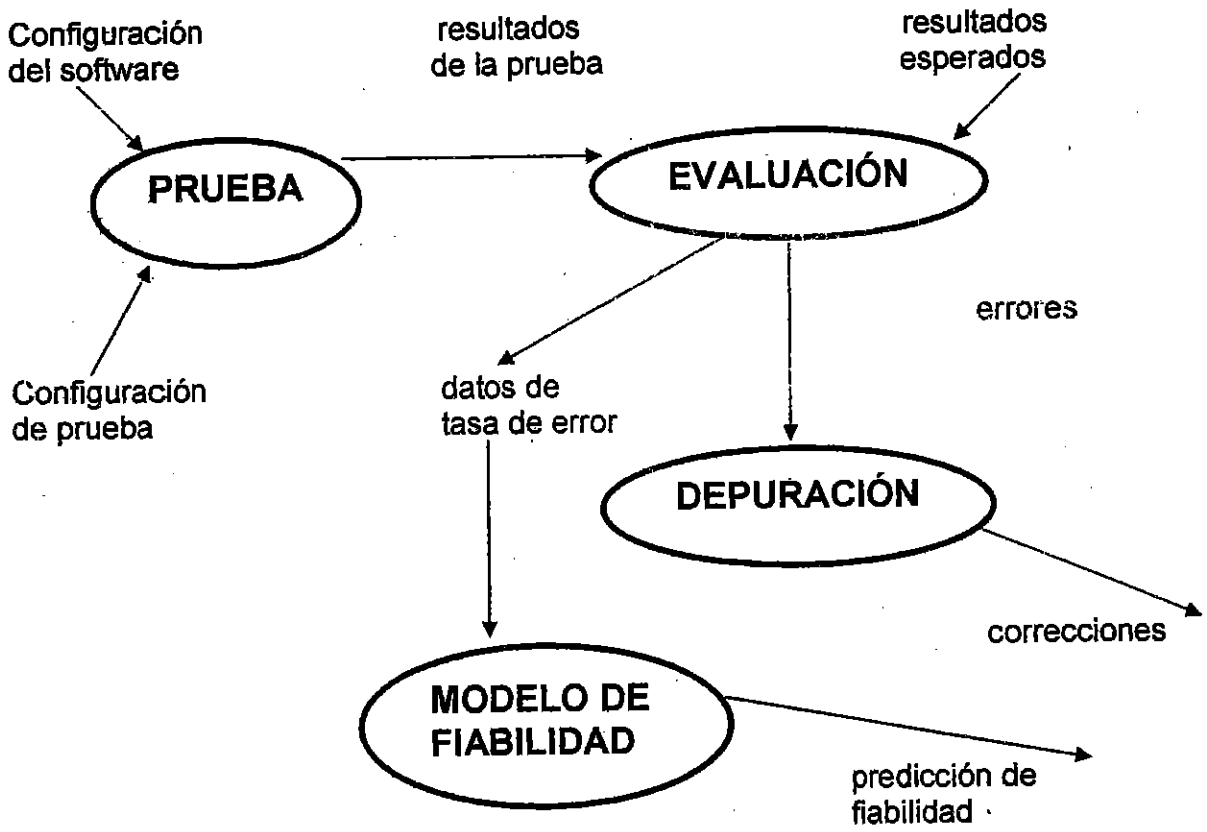
- La prueba es un proceso de ejecución de un programa que **intenta descubrir un error.**
- Un buen caso de prueba es aquél que tiene una **alta probabilidad de encontrar un error** que no había sido descubierto.
- **Una prueba tiene éxito si descubre un error** que no se había encontrado anteriormente.

Los datos que se van recogiendo a medida que se lleva a cabo la prueba proporcionan un buen indicador de la fiabilidad del software y, de alguna manera, indican la calidad del software como un todo.

> **La prueba no puede asegurar la ausencia de errores, sólo puede demostrar que existen defectos en el software.**

Flujo de Información de la Prueba

El flujo de información para la prueba sigue el siguiente esquema:



Obsérvese que las entradas al proceso de prueba son: **1) La configuración del software**, que incluye la especificación de requisitos, las especificaciones de diseño y el código fuente y **2) la configuración de prueba** que incluye un plan y procedimiento de prueba, algunas herramientas de prueba, casos de prueba y resultados esperados.

Si en el proceso de prueba se encuentran muchos errores serios, la calidad y fiabilidad del sistema queda en duda y se requieren más pruebas.

Si se está seguro de que las pruebas son adecuadas para descubrir errores y el proceso de prueba no los descubre, sin duda el usuario sí los descubrirá y el ingeniero deberá corregirlos durante la fase de mantenimiento.

Diseño de los casos de prueba

El diseño de casos de prueba puede requerir tanto o más esfuerzo como la etapa del diseño inicial del producto.

Los métodos de diseño de casos de prueba proporcionan **un enfoque sistemático a la prueba y un mecanismo de ayuda para asegurar la completitud de las pruebas** y conseguir una mayor probabilidad de descubrimiento de errores en el software.

Existen **dos tipos** de pruebas:

- **Pruebas de caja negra**

Conociendo la función específica para la que se ha diseñado el producto de software, se pueden llevar a cabo pruebas que demuestren que **cada función es completamente operativa.**

En el caso del software para computadoras las pruebas se llevan a cabo sobre la interfaz del software. Los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada, que se produce una salida correcta y se mantiene la integridad de la información externa (archivos de datos). Este tipo de prueba **no tiene mucho en cuenta la estructura lógica interna del software.**

- **Pruebas de caja blanca**

Conociendo el funcionamiento del producto se pueden desarrollar pruebas que **aseguren que la operación interna se ajusta a las especificaciones** y que todos los componentes internos se han probado en forma adecuada.

Este tipo de prueba se basa en un minucioso examen de los detalles procedimentales. Se comprueban los mecanismos lógicos del software ya que conjuntos específicos de casos de casos de prueba ejercitan lazos y condiciones. Se puede examinar el estado de cada programa en varios puntos para analizar si coincide con el estado esperado.

El único problema que se presenta es que **para grandes sistemas, la prueba exhaustiva es impracticable.**

Prueba de Caja Blanca

Este método de prueba usa la estructura de control del diseño procedimental para derivar casos de prueba que satisfagan las siguientes **condiciones:**

- Garanticen que se prueba, por lo menos una vez, todos los **caminos independientes** de control de cada módulo.
- Ejerciten por lo menos una vez todas las **decisiones lógicas** en sus versiones verdadera y falsa.
- Ejecuten todos los **lazos** en sus límites y con sus límites operacionales.
- Ejerciten las **estructuras internas de datos** para asegurar su validez.

Este tipo de prueba se usa para descubrir errores que muchas veces pasan inadvertidos a la prueba de caja negra.

Prueba de Caja Negra

Este método de prueba se centra en los requisitos funcionales del software ya que permite al ingeniero de software obtener un conjunto de condiciones de entrada que ejerciten todos los requisitos funcionales de un programa. Es un enfoque complementario a las prueba de caja blanca que permite descubrir otros **tipos de errores** tales como:

1. Funciones incorrectas o ausentes.
2. Errores de interfaz.
3. Errores de estructuras de datos o en accesos a bases de datos externas.
4. Errores de rendimiento.
5. Errores de inicialización o de finalización.

La prueba de caja negra, generalmente, se aplica en etapas posteriores a las pruebas de caja blanca.

Las pruebas de este tipo se diseñan para responder a las siguientes **preguntas**:

- ¿Cómo se prueba la validez funcional?.
- ¿Qué clase de entradas serán buenos casos de pruebas?.
- ¿Es el sistema particularmente sensible a ciertos valores de entrada?.

- ¿De qué forma están aislados los límites de una clase de datos?.
- ¿Qué volumen y nivel de datos tolerará el sistema?.
- ¿Qué efectos sobre la operación del sistema tendrán combinaciones específicas de datos?.
-

Conclusiones

El principal objetivo del diseño de casos de prueba es derivar un conjunto de pruebas que tengania mayor probabilidad de descubrir los defectos del software. Para ello se usan dos técnicas de diseño de casos de prueba: pruebas de caja blanca y pruebas de caja negra.

Las **pruebas de caja blanca** se centran en las estructuras de control del programa y los casos de prueba deben asegurar que se han probado por lo menos una vez todas las sentencias del programa y se han controlado todas las condiciones.

Este tipo de prueba se considera como una prueba a "pequeña escala" ya que típicamente es aplicada a pequeños componentes de programa. Por el contrario, la prueba de caja negra amplía en enfoque y se podría considerar como una ***"prueba a gran escala".***

Las **prueba de caja negra** son diseñadas para validar los requisitos funcionales sin fijarse en el funcionamiento interno de los programas. Se centran en el ámbito de información de un programa de forma que se proporcione una cobertura completa de prueba.

De todos modos, cuando el sistema se entrega al cliente que lo use, él se hará cargo de seguir llevando a cabo casos de prueba.

3. Estrategias de prueba de software

Una estrategia para probar el software **consiste en integrar las técnicas de diseño de casos de prueba** en una serie de pasos bien planificados que dan como resultado la correcta construcción del software. Además, proporciona una guía para el desarrollador del software, para el control de la calidad y para el usuario.

Obviamente, **cualquier estrategia de prueba debe incluir** la planificación de la prueba, el diseño de los casos de prueba, la ejecución de la prueba y la evaluación de los datos resultantes.

En otras palabras, **la prueba es un conjunto de actividades que se deben planificar y llevar a cabo sistemáticamente.**

En general, todas las **plantillas** para la prueba, que son un conjunto de pasos en los que se pueden situar las técnicas de diseño de casos de prueba, tienen las siguientes **características**:

- La prueba **comienza a nivel de módulo y trabaja hacia fuera,** hacia la integración de todo el sistema.
- **Diferentes técnicas de prueba son adecuadas en diferentes momentos.**
- La **prueba debe ser llevada a cabo** por el desarrollador y por un grupo de prueba independiente.
- La **prueba y la depuración** son actividades diferentes, pero la depuración debe incluir cualquier estrategia de prueba.

Una estrategia de prueba combina las **pruebas de bajo nivel**, que testean cada pequeño segmento de código para ver si se ha implementado correctamente, con **pruebas de alto nivel**, que demuestren la validez de las principales funciones del sistema frente a los requisitos del cliente.

Verificación y Validación

La prueba de software es parte de un concepto más amplio llamado verificación y validación.

- **Verificación** Conjunto de actividades que aseguran que el software implementa correctamente una función específica. **¿Estamos construyendo un producto correctamente?.**
- **Validación** Conjunto diferente de actividades que aseguran que el software construido se ajusta a los requisitos del cliente. **¿Estamos construyendo el producto correcto?.**

Los métodos de análisis, diseño y de implementación de software actúan para mejorar la calidad al proporcionar **técnicas uniformes y resultados predecibles.**

La prueba consiste en el último bastión desde el que se puede evaluar la calidad y corregir errores. Pero **la prueba no genera calidad**, si ésta no está presente en el software, tampoco lo estará al construir la prueba.

> **La calidad se incorpora al software durante el proceso de ingeniería, cuando se aplican adecuadamente los métodos y herramientas, y durante las revisiones formales efectivas.**

Organización para la prueba del software

En este momento se propone:

- El **desarrollador del software** es responsable de probar las unidades funcionales (módulos del programa), asegurándose que cada una lleva a cabo la función para la cual fue diseñada. Además se puede encargar de la prueba de Integración (paso de prueba que lleva a la construcción y prueba de la estructura total del sistema).

- Una vez que la arquitectura del software está completa entra a jugar un **grupo independiente de prueba**. Este grupo elimina el conflicto de interés presente cuando el constructor de software lo prueba. Trabajando estrechamente vinculados, el constructor y el grupo independiente de prueba, a lo largo del proyecto de software, aseguran que se realicen pruebas exhaustivas. Los errores deberán siempre ser corregidos por el desarrollador.

Una Estrategia para la Prueba del Software

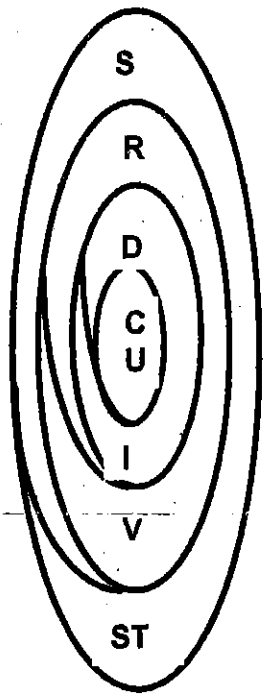
El proceso de ingeniería de software puede verse gráficamente en el siguiente **espiral:**

Ingeniería del Sistema

Requisitos

Diseño

Codificación



Prueba de Unidad

Prueba de Integración

Prueba de Validación

Prueba del Sistema

Inicialmente, la **Ingeniería de Software** define el papel del software y conduce el **análisis de requisitos** del software. En esta etapa se establece el campo de información, la funcionalidad, el comportamiento, el rendimiento, las restricciones y los criterios de validación del software. Moviéndonos hacia el centro del espiral llegamos al **diseño** y por último a la **codificación.**

Para desarrollar el software para computadoras damos vueltas en espiral a través de una serie de **líneas que disminuyen el nivel de abstracción** en cada vuelta.

La estrategia de prueba de software que se propone es **movernos hacia afuera del espiral.**

La **prueba de unidad** comienza en el vértice del espiral y se centra en cada unidad de software tal como está implementada en código fuente.

La prueba avanza, moviéndose hacia fuera del espiral, hasta llegar a la **prueba de integración**, donde el foco de atención es el diseño y la construcción de la arquitectura del software.

Dando otra vuelta hacia fuera del espiral, encontramos la **prueba de validación**, donde se validan los requisitos establecidos en el análisis de requisitos, comparándolos con el sistema que se ha construido.

Finalmente, se llega a la **prueba del sistema** donde se prueba todo el software y otros elementos del sistema como un todo.

Cada vuelta del espiral aumenta el alcance de la prueba.

Dentro del contexto de la Ingeniería de software, el procedimiento de prueba es en realidad una serie de **tres pasos que se llevan a cabo secuencialmente:**

1. Inicialmente la prueba se centra en cada módulo asegurándose de que funciona adecuadamente como unidad. Esta prueba de unidad hace un uso intensivo de la pruebas de caja blanca, ejercitando caminos específicos de la estructura de control del módulo para asegurar un alcance completo y una detección máxima de errores.

2. A continuación se integran los módulos para formar el paquete de software completo. La prueba de integración se dirige a todos los aspectos relacionados con la verificación y la construcción del sistema. Las técnicas que prevalecen son las de diseño de casos de prueba de caja negra,

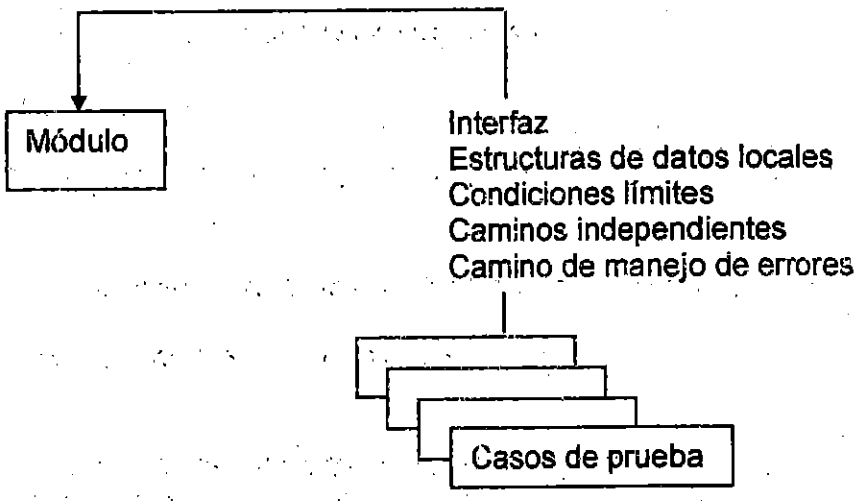
aunque se pueden llevar a cabo algunos casos de prueba de caja blanca para asegurar que se cubren los principales caminos de control.
Una vez construido el sistema se llevan a cabo un conjunto de pruebas de alto nivel. Se debe comprobar los criterios de validación establecidos durante el análisis de requisitos. La prueba de validación controla que el software satisfaga los requisitos funcionales, de comportamiento y de rendimiento establecidos. Se usan exclusivamente pruebas de caja negra.

3. El software una vez validado se combina con otros elementos tales como software, usuarios, bases de datos, etc. La prueba de Sistema verifica que cada elemento encaja de forma adecuada y que se alcanza la funcionalidad y rendimientos requeridos para el sistema en su totalidad.

Pruebas de Unidad

La prueba de unidad centra el proceso de verificación en la menor unidad de diseño, **el módulo**. Se usa como guía el modelo de **diseño detallado** y se prueban los caminos de control importantes con el fin de descubrir errores dentro del ámbito del módulo. Esta prueba está orientada a las pruebas de caja blanca y el paso de prueba se puede llevar a cabo en paralelo para múltiples módulos.

Las pruebas que se realizan como parte de la prueba de unidad se muestran en la siguiente figura:



Se prueba la **interfaz** del módulo para asegurar que la información fluye, desde y hasta el módulo. adecuadamente.

Se controlan las **estructuras de datos locales** para asegurar que los datos mantienen temporalmente su integridad durante todos los pasos del algoritmo.

Se prueban las **condiciones límites** para asegurar que el módulo funciona correctamente en los límites establecidos como restricciones de procesamiento.

Se ejercitan los **caminos independientes** o básicos de la estructura de control para asegurar que todas las sentencias del módulo se ejecutan al menos una vez.

Finalmente, se prueban todos los **caminos de manejo de errores**.

Obviamente, al iniciar cualquier prueba es preciso **probar el flujo de datos de la interfaz del módulo** ya que, si los datos no ingresan correctamente, las pruebas restantes no tienen sentido. Las siguientes preguntas deben ser satisfechas:

¿El número de parámetros de entrada es igual al número de argumentos?.
¿Coinciden los atributos de los parámetros y los argumentos?.
¿Coinciden los sistemas de unidades de parámetros y los argumentos?.
¿El número de argumentos transmitidos a los módulos es igual al número de parámetros?.
¿Son iguales los atributos de los argumentos transmitidos que los atributos de los parámetros?.
¿Son iguales sus sistemas de unidades?.
¿Son correctos el número, los atributos y el orden de los argumentos de las funciones incorporadas?.
¿Existen referencias a parámetros que no estén asociados con el punto de entrada local?.
¿Entran solamente parámetros alterados?.
¿Las definiciones de variables globales son consistentes entre módulos?.
¿Se pasan las restricciones como argumentos?.

Si además el módulo lleva a cabo **Entrada/Salida externa**, se deben satisfacer, adicionalmente las siguientes preguntas:

¿Son correctos los atributos de los archivos?.
¿Son correctas las sentencias de apertura?.
¿Cuadran las especificaciones de formato con las sentencias de E/S?.
¿Cuadra el tamaño del buffer con el tamaño del archivo?.
¿Se abren los archivos antes de usarlos?.
¿Se tienen en cuenta las condiciones de EOF?.
¿Se manejan los errores de E/S?.
¿Hay algún error textual en la información de salida?.

Ya que las **estructuras de datos locales** son una fuente potencial de errores se deben diseñar pruebas para descubrir errores de las siguientes **categorías:**

1. **Tipificación impropia o inconsistente.**
2. **Inicialización o valores implícitos erróneos.**
3. **Nombres de variables incorrectos (mal escritos o truncados).**
4. **Tipos de datos inconsistentes.**
5. **Excepciones de desborde, por arriba o por abajo, o de direccionamiento.**

Además de las estructuras de datos locales, durante la prueba de unidad, se debe comprobar el **impacto de los datos globales sobre el módulo.**

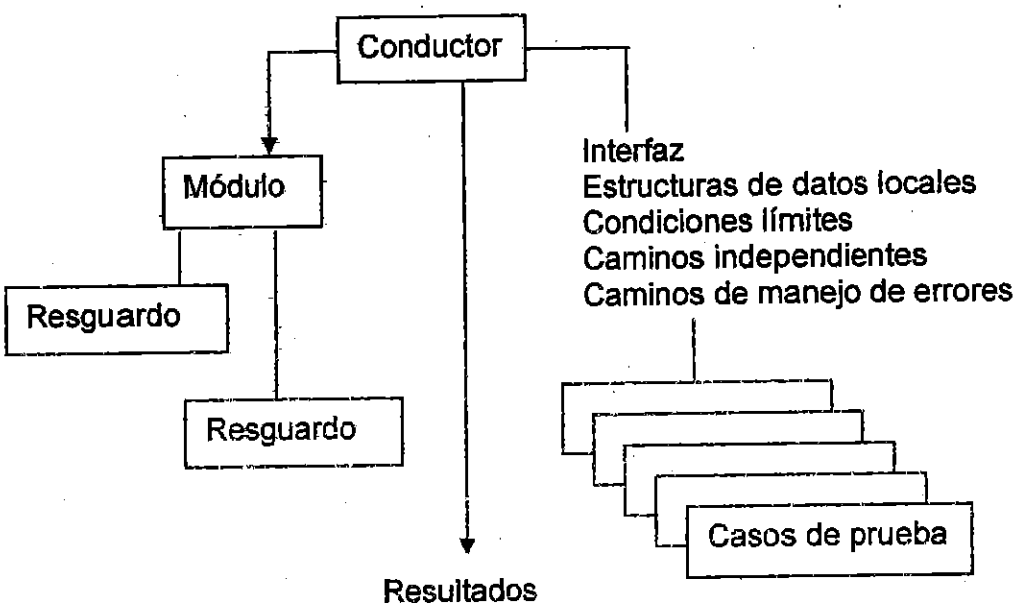
Se debe hacer una comprobación selectiva de los **caminos de ejecución**, diseñar casos de prueba para detectar errores debidos a **cálculos incorrectos,** comparaciones incorrectas o flujos de control inapropiados.

Las comparaciones y el flujo de control están fuertemente emparejados, por lo tanto, los casos de prueba deben descubrir errores tales como:

Comparaciones entre tipos de datos diferentes,
operadores lógicos o de precedencia incorrectos,
igualdad esperada cuando los errores de precisión la hacen poco probable,
variables o comparadores incorrectos,
finales de lazos inadecuados,
fallo de salida cuando se encuentra una iteración divergente,
variables de control de lazos modificadas de forma inadvertida o inadecuada,
etc.

La **prueba de límites** es la última tarea en el paso de prueba de unidad. Es muy importante ya que el software falla en condiciones límites, es decir, cuando se procesa el n-ésimo elemento de un arreglo n-dimensional, cuando se hace la i-ésima iteración en un lazo de i pasos o cuando se encuentran los valores máximo o mínimo permitidos. Los casos de prueba que ejerciten las estructuras de datos, el flujo de control y los valores de datos por encima de los máximos o por debajo de los mínimos son muy apropiados para descubrir errores.

El diseño de casos de prueba de unidad comienza una vez que se ha desarrollado, revisado y verificado en sus sintaxis el código fuente. **Se repasa la información de diseño para tener una guía para definir los casos de prueba** que tendrán una mayor probabilidad de descubrir errores de las categorías antes mencionadas. **Cada caso de prueba debe ir acompañado de un conjunto de resultados esperados.**



Ya que el módulo no es un programa independiente, para cada prueba de unidad se debe escribir un cierto software que conduzca **(DRIVER)** o resguarde **(RESGUARDO)** la prueba.

En la mayoría de los casos, un **DRIVER** no es más que un programa principal que acepte los casos de prueba, pase esos datos al módulo a probar, e imprima los resultados relevantes.

Por el contrario, los **RESGUARDOS** reemplazan módulos que son llamados por el módulo que se está probando. Un resguardo es un subprograma que usa la interfaz del módulo que lo contiene, lleva a cabo una mínima manipulación de datos, imprime una verificación de entrada y vuelve el control al módulo.

Ambos tipos de programas son **software adicional que no forma parte del producto del software final.** De acuerdo al grado de simplicidad que tenga el sistema pueden ser fáciles de escribir. Hay casos en que su uso se pospone hasta que se haga la prueba de integración.

La prueba de unidad se simplifica cuando se diseña un módulo con un **alto grado de cohesión**, ya que cuando el módulo se dirige a una función, se reduce el número de casos de prueba y los errores se pueden predecir y descubrir fácilmente.

Prueba de Integración

La prueba de integración es una **técnica para construir sistemáticamente la estructura del sistema de software** a la vez que se llevan a cabo pruebas para detectar errores asociados con la iteración. Obviamente, se espera que la estructura esté construida de acuerdo con el diseño.

Las razones por las que se producen errores al poner todos los módulos previamente probados, interactuando entre sí son:

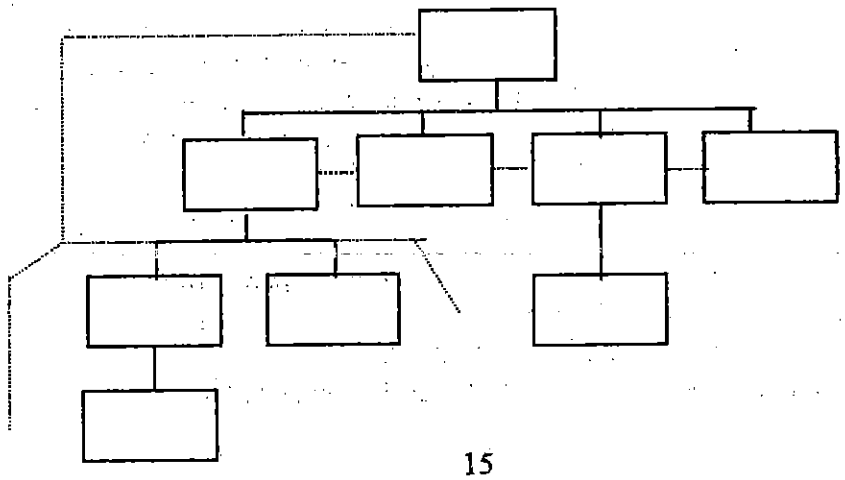
Se pueden perder datos en una interfaz,
Un módulo puede inadvertidamente afectar de una manera no deseada a otro,
Las subfunciones combinadas pueden no producir la función principal deseada,
Las estructuras de datos globales pueden presentar problemas,
El nivel de imprecisión aceptable para un módulo puede crecer hasta niveles inaceptables en el conjunto,
Etc.

Existe una tendencia a intentar una **integración no incremental** (enfoque Big-Bang) donde se combinan todos los módulos por anticipado y se prueba el programa en conjunto. En general esta metodología es caótica ya que se encuentran un gran número de errores, la corrección se hace muy difícil porque es muy complicado aislar la causa de los errores.

Por oposición, la **integración incremental** permite que el programa se construya y pruebe en pequeños segmentos fáciles de aislar y corregir y se puede aplicar un enfoque de prueba sistemático.

Integración Descendente

Se integran los módulos **moviéndose hacia abajo en la jerarquía de control**, comenzando por el programa principal. Los módulos subordinados se van incorporando en la estructura, ya sea por la forma primero-en –profundidad o bien la forma primero-en-ancho.



El proceso de integración se lleva a cabo en **5 pasos**:

1. Se usa el **módulo principal como conductor** de pruebas, disponiendo resguardos para todos los módulos directamente subordinados a él.

2. Se van **sustituyendo los resguardos subordinados** uno a uno por los módulos reales, de acuerdo a la forma de integración elegida.

3. Se llevan a cabo **pruebas** cada vez que se integra un módulo nuevo.

4. Tras terminar cada conjunto de pruebas, **se reemplaza otro resguardo** por el módulo real.

5. Se hace la **prueba de regresión**, es decir, todas o algunas de las pruebas anteriores, para asegurar que no se han producido nuevos errores.

El proceso se repite y continúa hasta que se haya construido la estructura del programa completo.

Integración Ascendente

Esta prueba comienza con la construcción y prueba de los **módulos atómicos** (módulos de los niveles más bajos de la estructura del programa). Dado que los módulos se integran de abajo hacia arriba, el procesamiento requerido de los módulos subordinados está siempre disponible y no se hacen necesarios los resguardos.

La estrategia de integración se implementa mediante los siguientes pasos:

1. Se combinan los módulos de bajo nivel en grupos llamados **construcciones** que realizan una subfunción específica.
2. Se escribe un **conductor** (programa de control de prueba) para coordinar la entrada y salida de casos de prueba.
3. **Se prueba el grupo.**
4. **Se eliminan los conductores y se combinan los grupos** moviendose hacia arriba en la estructura del programa.

3.5.1 Ventajas y Desventajas

La principal **desventaja del enfoque descendente** es la necesidad de resguardos y la dificultad de prueba asociada a ellos. La principal ventaja es que se pueden probar de antemano las principales funciones de control.

La mayor **desventaja de la integración ascendente** es que el programa como entidad no existe hasta que se ha añadido el último módulo. Pero permite una mayor facilidad de diseño de casos de prueba y no hay que escribir resguardos.

En general, el mejor compromiso puede ser un planeamiento combinado, llamado **prueba sándwich**, que use el enfoque descendente para los niveles superiores de la estructura del programa y el enfoque ascendente para los niveles subordinados.

A medida que progrese la prueba, su encargado debe identificar los **módulos críticos**, que son los que tienen una o más de las siguientes características:

1) Está dirigido a varios requisitos del software.
2) Tiene un mayor nivel de control (está alto en la estructura).
3) Es complejo o propenso a errores.
4) Tiene requisitos de rendimiento muy definidos.

Los módulos críticos deben probarse lo antes posible, y la prueba de regresión debe centrarse en ellos.

Prueba de Validación

Una vez que el software está completamente ensamblado como un paquete, que se han encontrado y corregido los errores de interfaz, puede comenzar la **prueba de validación.**

> **La validación se logra cuando el software funciona de acuerdo con las expectativas del cliente, definidas en las especificaciones de requisitos.**

La validación de software se consigue mediante **una prueba de caja negra** que demuestran la conformidad con los requisitos. Se hace un plan de pruebas y se usa un procedimiento de prueba para definir los casos de prueba específicos que se usarán para demostrar la conformidad con los requisitos. Ambos estarán diseñados para asegurar que se satisfacen con los requisitos funcionales, que se alcanzan los requisitos de rendimiento, que la documentación es correcta e inteligible y que alcanzan los requisitos de portabilidad, compatibilidad, recuperaciones de errores, facilidad de mantenimiento, etc.

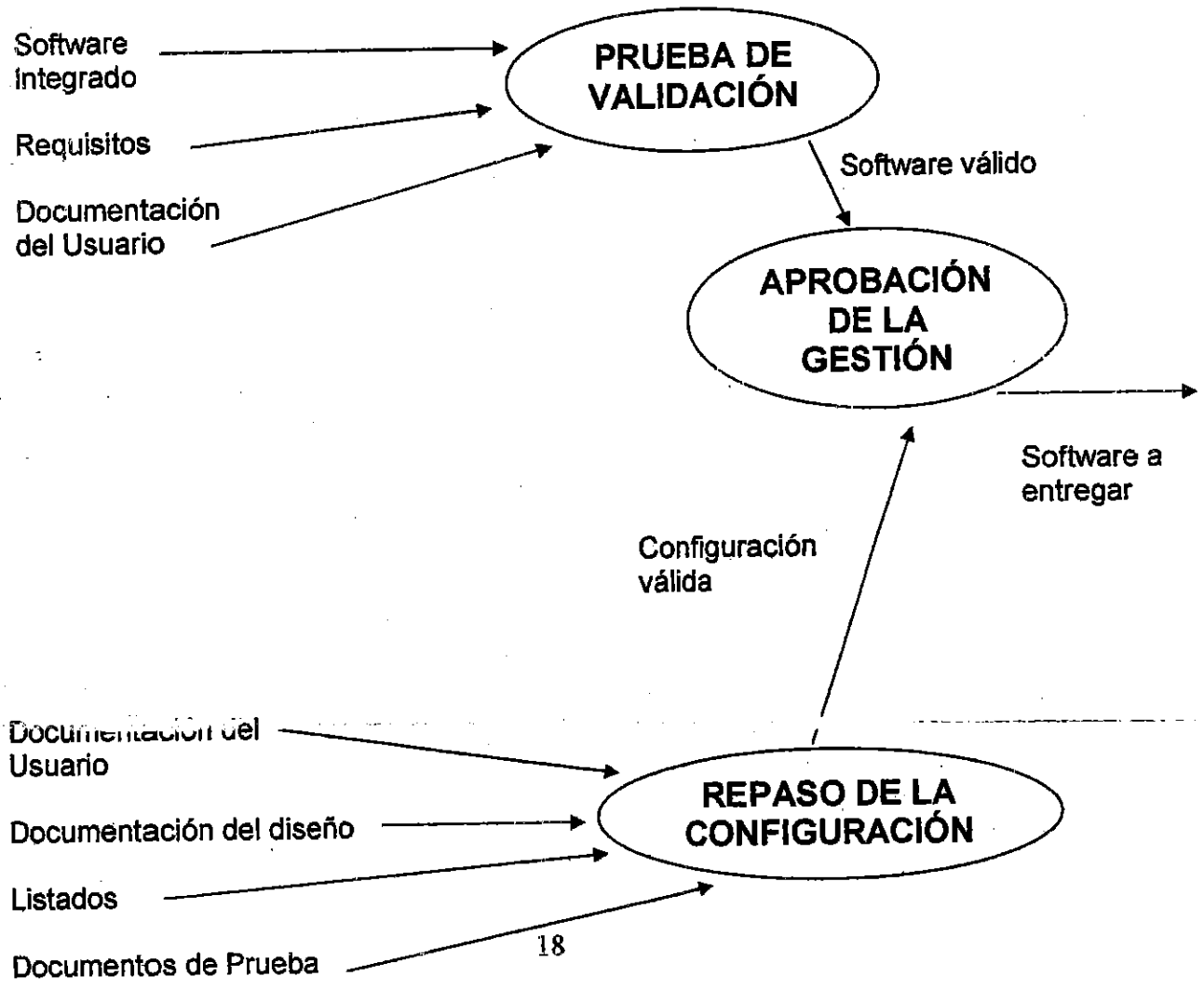
Una vez que se procede con cada caso de prueba de validación puede darse **una de dos condiciones:**

1) Las características de funcionamiento o rendimiento están de acuerdo a las especificaciones y **son aceptables**, o

2) Se descubre una desviación de las especificaciones y se crea una **lista de deficiencias**. Estos errores generalmente se corrigen al terminar el plan negociando con el cliente un método para resolverlos.

Un elemento importante de la prueba de validación es el **repaso de la configuración** (a veces llamado **auditoría**) que intenta asegurar que todos los elementos de la configuración de software se han desarrollado adecuadamente, están catalogados y tienen suficiente detalle para facilitar el mantenimiento.

Esta etapa o proceso se ilustra en la siguiente figura:



Pruebas Alfa y Beta

Cuando se construye software a medida para un cliente, se llevan a cabo una serie de **pruebas de aceptación** para que él mismo valide todos los requisitos. Esto se debe a que es imposible prever cómo el cliente usará realmente el programa debido a errores de interpretación delas instrucciones de uso, en el sentido, por ejemplo, de que una salida puede ser muy clara para quien realiza la prueba pero ininteligible para un usuario normal.

Una prueba de aceptación puede ir desde el informal paso de prueba hasta la ejecución sistemática de una serie de prueba bien planificadas.

La **prueba Alfa** es conducida por el cliente en el lugar de desarrollo, con el desarrollador presente para registrar errores y problemas de uso. Es decir, la prueba Alfa se lleva a cabo en un **ambiente controlado.**

La **prueba Beta** se lleva a cabo en ambientes del cliente y es conducida por el usuario final del sistema. El desarrollador normalmente no está presente, ya que se trata de una aplicación en vivo en un **entorno no controlado** por el equipo de desarrollo. Los resultados, anotados, de esta prueba llevan a modificaciones para producir un producto para toda la base de clientes.

Prueba del Sistema

La prueba del Sistema está formada por una serie de pruebas diferentes cuyo propósito primordial es **ejercitar profundamente al sistema**. Todas estas pruebas trabajan para verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas.

Prueba de Recuperación

El tiempo de recuperación ante los fallos y la tolerancia del sistema a los mismos es muy importante en los sistemas de software. En general, esto de alguna manera nos dice que los fallos de procesamiento no deben provocar el cese en el funcionamiento del software.

Una **prueba de recuperación** fuerza el fallo del software de alguna manera y verifica que la recuperación se lleve a cabo adecuadamente.

Si se ha previsto una **recuperación automática**, es decir que el propio sistema la lleva a cabo, se debe controlar la correctitud de los datos y el re-arranque.

Si, por el contrario, la recuperación requiere de la participación humana, se evalúan los tiempos medios de reparación para ver si están dentro de los límites aceptables.

Prueba de Seguridad

Cualquier sistema computacional que maneje información sensible es objeto de **actividades piratas**, que intentan entrar en los sistemas para obtener ganancias personales ilícitas.

La **prueba de seguridad** intenta verificar que los mecanismos de protección incorporados al sistema lo protegerán de las entradas o accesos impropios.

Durante la prueba, el encargado de la misma debe intentar hacerse con las claves de acceso por cualquier medio externo al oficio, puede atacar al sistema con software diseñado para romper las claves, debe bloquear el sistema negando el servicio a otras personas, debe producir a propósito errores del sistema, intentando entrar durante la recuperación, etc.

Con suficiente tiempo y recursos, toda buena prueba de seguridad finalmente accede al sistema. El rol del diseñador del sistema es hacer que el costo de entrar sea mayor que el valor de la información obtenida al hacerlo.

Prueba de Resistencia

Las pruebas de resistencia están diseñadas para enfrentar al sistema con **situaciones anormales**. El responsable de la prueba debe preguntarse, ¿a qué potencia se puede ponerlo a trabajar antes de que falle?.

Durante la prueba de resistencia se hace que el **sistema demande recursos** en cantidad, frecuencia o volúmenes anormales, y se estudia como reacciona el sistema ante estas anormalidades.

Prueba de Rendimiento

La prueba de rendimiento se diseña para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado.

Este tipo de prueba se ha venido haciendo durante todos los pasos del proceso de prueba, pero solamente se puede asegurar el rendimiento del sistema cuando están integrados todos los elementos del mismo.

4. El arte de la Depuración

La depuración aparece como consecuencia de una prueba efectiva ya que, si un caso de prueba descubre un error, **la depuración es el proceso que resulta en la eliminación del error.**

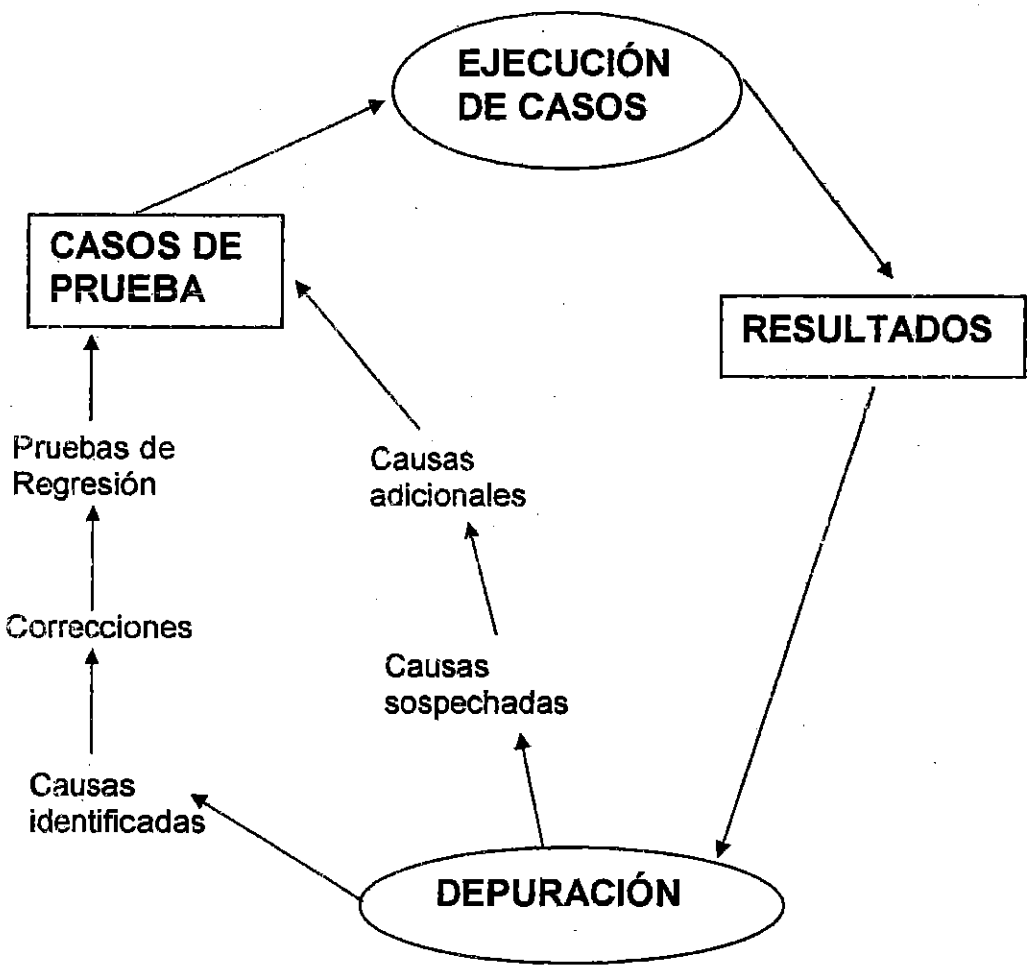
Aunque la depuración puede y debe ser un proceso ordenado, es el único proceso que aún conserva características que lo definen como artístico ya que es un **proceso mental que conecta un síntoma con una causa.**

El Proceso de Depuración

La depuración no es una tarea de prueba, pero sí **es una consecuencia de la prueba.**

En la siguiente figura se muestra como el proceso de depuración comienza con la ejecución de un caso de prueba, cuando se evalúan los resultados y no se corresponden con los esperados.

> **Estos datos que no corresponden, en muchos casos, son un síntoma de una causa que permanece oculta. La depuración se encarga de buscar esa causa, llevando así a la corrección del error.**



Si no se puede encontrar la causa, se diseñan casos de prueba que ayuden a validar las sospechas de donde está hasta que se la encuentra, se corrige y se eliminan los errores.

La razón del porqué de la depuración se dice que está más asociada a atributos personales que a una metodología, está relacionada con las características de los errores que se producen:

- El síntoma y la causa pueden ser remotos entre sí.
- El síntoma puede desaparecer temporalmente al corregir otro error.
- El síntoma puede producirse por una causa que no sea un error (redondeos).
- El error puede ser humano y de difícil detección.

- El síntoma puede ser resultado de problemas de tiempo y no de procesamiento.
- Puede ser difícil reproducir en forma precisa las condiciones de entrada (en un sistema de tiempo real el orden de la entrada no está determinado).
- El síntoma puede aparecer en forma intermitente.
- El síntoma puede producirse debido a causas que se distribuyen por un serie de tareas ejecutándose en diferentes procesadores.

En general, existen **tres enfoques** para proponer la depuración:

- **Fuerza Bruta:** Se ponen controles (escrituras) en los programas hasta que alguien logra detectar el error. Es el enfoque más común y menos eficiente.

- **Vuelta Atrás:** Se usa con éxito en pequeños programas, recorriendo manualmente el código hacia atrás, partiendo desde donde se detectó el síntoma, hasta que se llega a ubicar el error.

- **Eliminación de causas:** Se usa la inducción o deducción particionando en forma binaria el dominio de los datos relacionados con la ocurrencia del error, con el fin de aislar las posibles causas. Se llega a una hipótesis de causa y se usan los datos anteriores para probar o revocar esa hipótesis. Si la hipótesis es prometedora, se refinan los datos a fin de intentar aislar el error.

5. Pruebas Orientadas a Objetos

Ya se vio que el objetivo de la realización de pruebas es encontrar el mayor número posible de errores con una cantidad de esfuerzo racional a lo largo de un espacio de tiempo realista.

Aunque el *objetivo* se mantiene para el software OO, la naturaleza de los programas OO cambia la estrategia y tácticas de prueba.

Para probar adecuadamente los sistemas OO deben hacerse tres cosas:

(1) La definición de las pruebas debe ampliarse para incluir técnicas de detección de errores aplicados a los modelos de DOO y AOO.
(2) La estrategia para las pruebas de unidad e integración debe cambiar significativamente.
(3) El diseño de casos de prueba debe tener en cuenta las características propias del software OO.

La construcción del software OO comienza con la creación de modelos de análisis y diseño. Ya que el paradigma de la ingeniería del software OO es de naturaleza evolutiva, estos modelos comienzan con representaciones informales de requisitos del sistema y evolucionan hacia un modelo de clases detallado, conexiones y relaciones entre clases, diseño del sistema y asignaciones y diseño de objetos (se incorpora un modelo de conectividad entre objetos vía mensajes). Los modelos pueden probarse para descubrir errores antes de que se propaguen en la próxima iteración.

Todos los modelos orientados a objetos deberán probar su correctitud, completitud y consistencia dentro del contexto de la sintaxis del modelo, semántica y pragmática.

5.1 Modelos de pruebas AOO y DOO

Como los modelos de análisis y diseño no se pueden ejecutar, no se pueden probar en el sentido convencional. Pero sí se pueden usar las revisiones técnicas formales para examinar la correctitud y consistencia de ambos modelos.

La correctitud sintáctica de un modelo se juzga el uso apropiado de la simbología y las convenciones propias del modelo de análisis y diseño escogido para el proyecto.

La correctitud semántica de un modelo se juzga basándose en la conformidad del modelo con el dominio del problema del mundo real. Si el modelo refleja el mundo real de manera exacta entonces está semánticamente correcto (a un nivel de detalle apropiado a la etapa de desarrollo en la cual se revisa el

modelo). Para determinar esta correctitud, el modelo se presenta a los expertos del dominio, quienes examinarían las definiciones de clases y jerarquías en busca de omisiones y ambigüedades.

Se evalúan las relaciones de clases (conexiones de instancias), a veces rastreando o siguiendo los modelos contra escenarios de uso del mundo real, para determinar si reflejan exactamente las conexiones de objetos del mundo real.

La consistencia de los modelos de AOO y DOO puede juzgarse a través de una consideración de las relaciones entre entidades en el modelo. Un modelo inconsistente tiene representaciones que pueden no estar correctamente reflejadas en otras partes del modelo.

Debe examinarse cada clase y sus conexiones a otras clases.

Para facilitar esta actividad se usa el modelo CRC y el diagrama objeto – relación.
El diseño del sistema se revisa a través del examen del modelo objeto – comportamiento desarrollado durante el AOO.

5.2 Estrategias de pruebas Orientadas Objetos

5.2.1 Pruebas de Unidad en el contexto OO

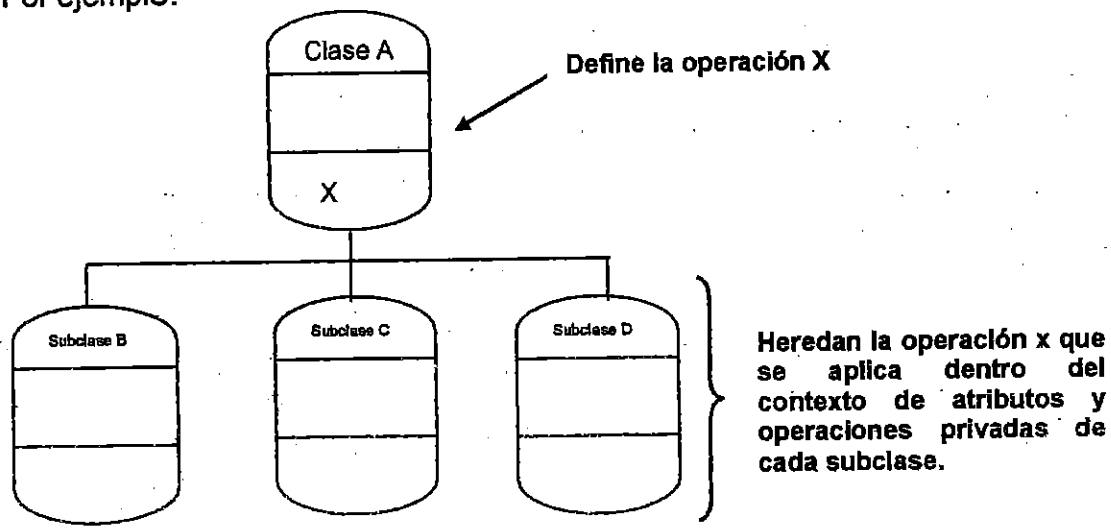
Ya que al considerar software OO cambia el concepto de unidad, el significado de la prueba de unidad cambia drásticamente.

El encapsulamiento dirige la definición de clases y objetos.
Cada clase o instancia de clase (objeto) empaqueta atributos y operaciones que manipulan esos datos.

En lugar de módulos individuales, la menor unidad a probar es la clase u objeto encapsulado. Una clase puede contener un cierto número de operaciones y una operación particular puede existir como parte de un cierto número de clases.

Una operación no se puede probar con detalle aisladamente sino como parte de una clase.

Por ejemplo:



Es necesario probar la operación X en el contexto de cada subclase

La prueba de clases para el software OO es equivalente a la prueba de unidad para el software convencional. Mientras la prueba de unidad se centra en el detalle algorítmico del módulo y los datos que fluyen a lo largo de su interfaz, la prueba de clases para software OO está dirigida por las operaciones encapsuladas en la clase y el estado del comportamiento

5.2.2 Prueba de integración en el contexto OO

Ya que el software OO no tiene estructura de control jerárquica las estrategias convencionales de integración ascendente y descendente tienen poco sentido. Más aún, la integración una a una de las operaciones de una clase es a menudo imposible debido a las **interacciones directas e indirectas de las componentes que conforman la clase.**

Existen dos estrategias diferentes para pruebas de integración en sistemas OO:

- **Pruebas basadas en hilos:** integra el conjunto de clases necesario para responder a una entrada o evento del sistema
- **Pruebas basadas en uso:** comienza la construcción del sistema probando aquellas clases (**independientes**) que usan muy pocas, o ninguna, clase servidor. Luego se prueban las **clases dependientes**, que usan las clases independientes.

La **prueba de agrupamiento** es uno de los pasos de la prueba de integración del software OO. Se ejercita un conjunto de clases colaboradoras a través del diseño de casos de prueba que intentan descubrir errores en las colaboraciones.

5.2.3. Prueba de Validación en el contexto OO

En el nivel de validación o del sistema, los detalles de conexiones de clase desaparecen. La validación del software OO, al igual que para el software convencional, se centra en las acciones visibles del usuario y las salidas del sistema que éste reconoce.

El ejecutor de la prueba debe basarse en los casos de uso que forman parte del modelos de análisis y que brinda un escenario que posee una alta probabilidad con errores encubiertos en los requisitos de interacción del cliente.

Para dirigir las pruebas de validación pueden usarse métodos de prueba de caja negra. También se pueden derivar casos de prueba a partir del modelo *objeto – comportamiento* y del diagrama de eventos como parte del AOO.

CRISIS CRÓNICA DE LA PROGRAMACIÓN

a) Por cada 6 nuevos sistemas de software de gran tamaño que entran en servicio, 2 quedan cancelados.
b) Los de tamaño medio suelen consumir 1 ½ veces el tiempo previsto.
c) ¾ partes de todos los sistemas de gran tamaño software "fracasos operativos (no funcionan como se quería o no se usan).

> **Ingeniería de Software: La aplicación de métodos sistemáticos, disciplinados y cuantificables al desarrollo, funcionamiento y mantenimiento del software.**

Hay esperanzas:

a) La intuición va cediendo paso al análisis.
b) Los programadores comienzan a utilizar medidas cuantitativas de la calidad del software que producen para mejorar la forma en que los producen.
c) Los métodos para la expresión algebraica de los diseños de programas consolidan los fundamentos matemáticos de la programación.
d) Existe una nueva generación de profesionales de software competentes.
e) Muchas personas de la industria centran su atención en la invención de tecnologías y estructuras mercantiles necesarias para soportar las partes del software que sean intercambiables y reutilizables.

Los sistemas distribuidos pueden consistir en un conjunto muy grande de puntos individuales interconectados, en lo que cabe que se produzcan fallas y muchos de los cuales no están identificados de antemano.

> La complejidad de estos sistemas plantean un problema de primera magnitud.

El desafío de la complejidad es grande y va en aumento. Cuando un sistema se vuelve tan complejo que ningún gerente lo comprende individualmente por completo, los procesos tradicionales de desarrollo se vienen abajo.

> **Empresas líderes de la industria han comprendido bastante sobre cómo medir cuantitativa y consistentemente, el caos de sus procesos de desarrollo, la densidad de errores de sus productos y el estancamiento de la productividad de sus programadores.**

CMM: Capability Maturity Model aplica una escala de 5 niveles, desde el caos (nivel 1) hasta una buena gestión (nivel 5).

75 % de nivel 1. Carecen de procesos formales, no miden lo que hacen y no tienen forma de saber si van por mal camino o han descarrilado del todo.

Nivel 5 → Equipo de programación de Motorola en Bangalore, India.
→ Proyecto de software de a bordo para el trasbordados espacial, realizado por Loral (antes de IBM).

Los "bugs" que provocan efectos desvastadores son casi siempre deslices cometidos en el diseño inicial que llegan intactos hasta el producto definitivo.

Enfoque posteriori y de fuerza bruta para eliminar los defectos:	- Versión Beta	Sistema eficaz pero muy oneroso, ineficiente y poco práctico.

Los investigadores formulan diversas maneras para atacar los errores tempranamente e incluso impedir que ocurran.

El software es algo que, más que construirse se cultiva.

En un primer paso, se hilvana rápidamente un prototipo merced a componentes estándares de interfases gráficas. Este puede servir para aclarar malentendidos entre el programador y su cliente antes de empezar los cimientos lógicos.

Pero los prototipos sirven de poco en la detección de inconsistencias lógicas del diseño ya que sólo remedan el aspecto externo de su comportamiento.

Valiéndose de la matemática discreta: teoría de conjuntos y del cálculo de predicados, los científicos de la computación se las han arreglado para traducir especificaciones y programas al lenguaje matemático donde pueden analizarse con los instrumentos teóricos llamados **métodos formales**.

La seguridad es una preocupación obvia
Los test funcionales siguen siendo necesarios por dos razones:
a) Los programadores cometen errores en las demostraciones y
b) Los métodos formales sólo pueden garantizar que el software cumple con su especificación, no que sepan manejarse con las sorpresas que depara el mundo real.

El **enfoque de sala limpia** (clean-room approach) trata de conjugar las notaciones formales, las demostraciones de validez y los controles estadísticos de calidad con un enfoque evolutivo del desarrollo de software. Trata de aplicar consistentemente técnicas rigurosas de ingeniería para fabricar productos que funcionen perfectamente la primera vez. **Los programadores desarrollan el sistema de a una función por vez y se aseguran de certificar la calidad de cada una antes de integrarla a la arquitectura**. Este tipo de software requiere una nueva metodología de testeo.

En un proceso de sala limpia, los programadores tratan de asignar una probabilidad a cada camino de ejecución que los usuarios puedan tomar (correcto o incorrecto). Así pueden derivar casos de prueba de datos estadísticos, de forma que las rutas más frecuentemente utilizadas se analicen más profundamente. Luego se hace funcionar el programa en cada uno de estos casos de prueba y se cronometra cuánto tarda en fallar. Estos datos (tiempos) se cargan en un modelo matemático que calcula la confiabilidad del programa, a la manera más puramente ingenieril.

Nadie sabe cuán productivos son quienes desarrollan software por tres razones:

- Menos del 10% de las compañías norteamericanas miden de forma coherente y sistemática la productividad de sus programadores.
- En el sector no se han establecido una unidad de medida estándar y útil.
- Las diferencias individuales en la productividad anulan los efectos de mejoras tecnológicas o de procesos

La existencia de la computación requiere una rama experimental destinada a separar los resultados generales de los meramente accidentales.

Una vez estandarizadas, las partes del software pueden reutilizarse en diferentes escalas. El desafío consiste en romper los lazos que ligan los programas a computadores específicos y a otros programas.

Se estudia, entre otras cosas, un lenguaje común que podría servir para describir partes de software; programas capaces de reformar los componentes, adaptándolos a un ambiente cualquiera; componentes provistos de opciones múltiples, que el usuario podría activar o no.

Lo que quieren los compradores es pagar por el componente una vez y luego hacer copias gratis.

La combinación de control industrial de procesos, herramientas técnicamente avanzadas y partes intercambiables promete transformar no sólo la forma de realizar la programación sino también a los encargados de efectuarla.

1. Ingeniería de Software

1.1. Introducción

Ya que actualmente las economías personales, empresariales, nacionales e internacionales dependen cada vez más de las computadoras y sus sistemas de software, es que **la práctica de la Ingeniería de Software tiene por objeto la construcción de grandes y complejos sistemas en forma rentable**.

Los problemas que se presentan en la construcción de grandes sistemas de software **no son simples versiones a gran escala** de los problemas de escribir pequeños programas en computadoras.

La complejidad de los **programas pequeños** es tal que una persona puede comprender con facilidad y retener en su mente todos los detalles de diseño y construcción. Las especificaciones pueden ser informales y el efecto de las modificaciones puede ser evidente de inmediato. Por otro lado, los **grandes sistemas** son tan complejos que resulta imposible para cualquier individuo recordar los detalles de cada aspecto del proyecto. Se necesitan técnicas más formales de especificación y diseño: debe documentarse adecuadamente cada etapa del proyecto y es esencial una cuidadosa administración.

El término **Ingeniería de Software** se introduce a fines de la década del 60, al producirse la llamada "Crisis del Software" a causa de la aparición de Hardware de 3ª generación. Estas nuevas máquinas eran de una capacidad muy superior a las máquinas más potentes de 2ª generación y su potencia hizo posible aplicaciones hasta entonces irrealizables.

Las primeras experiencias en la construcción de grandes sistemas de software mostraron que las técnicas de desarrollo hasta entonces conocidas eran inadecuadas. Entonces se presentó la urgente necesidad de **nuevas técnicas** y **metodologías** que permitieran controlar la complejidad inherente a los grandes sistemas de software.

Ingeniería de Software = Programming in the Large

Entonces, convenimos en definir:

Ingeniería de Software Trata de la construcción de grandes sistemas que no puede manejar un único individuo. Usa principios metodológicos de la Ingeniería para el desarrollo de sistemas, es decir, sistematiza el desarrollo de sistemas. Adquiere el nivel de una disciplina. Consta de aspectos técnicos y aspectos no técnicos.

El **Ingeniero de Software** debe tener profundos conocimientos de las técnicas de computación y debe poder comunicarse en forma oral y escrita. Debe

comprender los problemas de administración de proyectos relacionados con la producción de software y debe poder apreciar los problemas de los usuarios del software cuando no lo entienden.

La Ingeniería de Software **abarca**:

Métodos Incluyen el planeamiento del sistema, análisis de requisitos, diseño de estructura de datos y procedimientos algorítmicos, codificación y testeo.

Herramientas Son soportes automatizados y semiautomatizados para los métodos. Cuando este soporte es un sistema integrado para análisis, diseño, codificación y testeo recibe el nombre de herramienta CASE (COMPUTER AIDED SOFTWARE ENGINEERING).

Procedimientos Definen la secuencia en que se aplican los métodos, los derivables y los controles.

CONCEPTO DE SOFTWARE

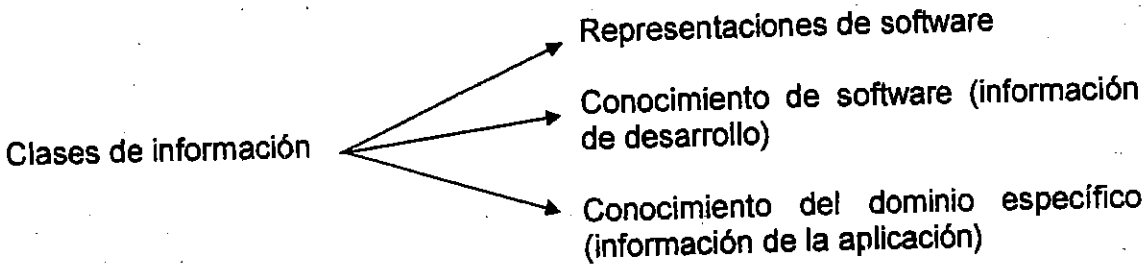
El software es:

- Alma y cerebro de una computadora.
- Corporización de los fines de un sistema.
- El conocimiento adquirido acerca de un área de aplicación.
- Conjunto de programas y datos necesarios para convertir una computadora (de propósito general) en una máquina de propósito específico para una aplicación particular.
- Documentación producida durante el desarrollo de un sistema software-intensivo.

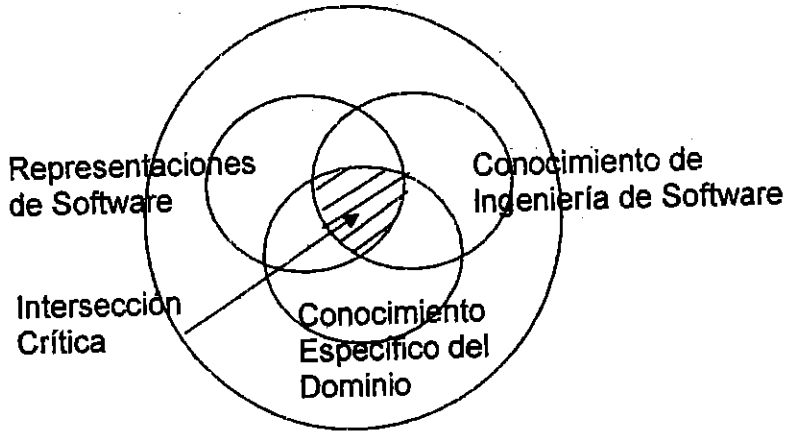
Pensar que el software es sólo programas genera problemas: Ej. medir la productividad por líneas de código producidas por unidad de tiempo (capacidad de generar código ≡ productividad en la construcción del sistema). Así se condiciona el ambiente para producir código que lleva a montañas de código que no se pueden integrar para trabajar como un sistema y a la construcción de sistemas técnicamente correctos que no satisfacen las necesidades de los usuarios.

Todo lo que es el software son los aspectos de la información. El **software no sólo son programas ejecutables ya que se excluye toda información relevante.**

La clave de un desarrollo exitoso es no perder o alterar información introduciendo errores.



Información en el ambiente de desarrollo



Representaciones del Software

Cualquier información que en forma directa representa un eventual conjunto de programas y los datos asociados

- Programas
- Diseños detallados
- Diseño de arquitectura (Diagramas de Estructuras)
- Especificaciones escritas en lenguaje formal
- Requisitos del sistema expresados en una combinación de notaciones.
- Etc.

Conocimiento de la Ingeniería de Software

Toda la información relativa al desarrollo en general (ej.: cómo usar un método específico de diseño) o relativa a un desarrollo particular (ej.: programa de testeo en un proyecto).

- Información relativa al proyecto.
- Información sobre la tecnología de software (métodos, conceptos, técnicas).
- Conocimiento acerca de sistemas similares.
- Información detallada relativa a la identificación y solución de problemas técnicos del sistema en desarrollo.

Conocimiento del dominio específico

Es esencial para la creación del software. Descubrirlo y ponerlo en práctica en forma útil es la esfera de un especialista en el área de aplicación.

- Conocimiento del proceso específico a ser controlado.
- Reglas de contabilidad.
- Procedimientos para actualizar y cambiar los registros de los empleados.
- Etc.

Formas que toma el software

- Programas en lenguaje de máquina.
- Programas en lenguaje de alto nivel.
- Especificaciones.
- Necesidades.
- Requerimientos.
- Diseños arquitectónicos.
- Diseños detallados.
- Formatos de datos.
- Colecciones de programas.
- Programas a testear.
- Programas terminados.
- Sistemas en uso para producción

Otras formas:

- Análisis de Requisitos
- Documentación del usuario.
- Documentación de mantenimiento.
- Pedidos de cambio.
- Especificaciones de modificaciones.
- Informes de errores.
- Mediciones de performance.

El software es tanto un producto como un objeto, esto es: conocimiento empaquetado.

Los programas (final de una cadena de representaciones que llamamos software) contienen conocimiento. Una de las principales razones de la **reusabilidad** es no perder este conocimiento.

Software como conocimiento – Balancear actividades de análisis y la construcción de programas. Es la visión más idónea para entender la naturaleza esencial de la actividad.

Software como producto – Se vincula con la organización del desarrollo de software.

Software como una serie de transformaciones – Provisión de herramientas para ayudar al desarrollo de software.

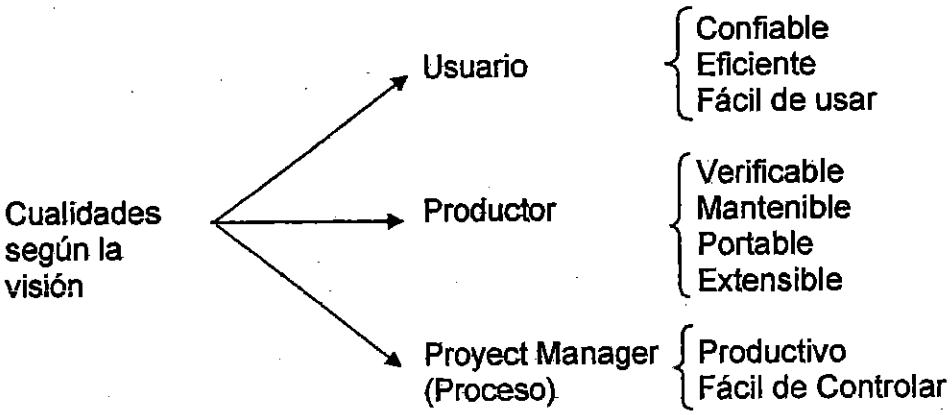
Productos de Software

Programas de Computadoras	+ Procedimientos +	Documentación Asociada	+	Datos de la operación del sistema	=Conocimiento Acumulado

CARACTERÍSTICAS DEL SOFTWARE

Características Conceptuales

- Objetivo de la Ingeniería: Construcción de productos. Para la Ingeniería de software, el producto son los sistemas de software.
- El software es maleable.
- Se cree que cambios en el software son fáciles.
- Un cambio en el software debe verse como un cambio en el diseño más que en el código.
- Su producción es humano-intensiva: requiere más ingeniería que manufactura. El proceso de producción de software se vincula más con el diseño e implementación que con la manufactura.
- En la Ingeniería de Software, a diferencia de lo tradicional, el ingeniero dispone de herramientas para describir el producto que no son distintas del producto.
- A menudo, las cualidades del producto están entremezcladas en especificaciones con las cualidades del diseño.



Cualidades externas: Visibles para los usuarios.
Cualidades internas: Sólo visible para los desarrolladores.
Existe una fuerte relación entre ellas: La cualidad interna de al verificabilidad se requiere para alcanzar la cualidad externa de la confiabilidad.

Confiabilidad ⇨ verificabilidad

Cualidades del software Procesos y Productos

- Corrección funcional.
- Confiabilidad.
- Robustez.
- Performance.
- "Amistosidad".
- Verificabilidad.
- Mantenibilidad.
- Reusabilidad.
- Portabilidad.
- Comprensibilidad.
- Interoperatividad.
- Productividad.
- Oportunidad.
- Visibilidad.

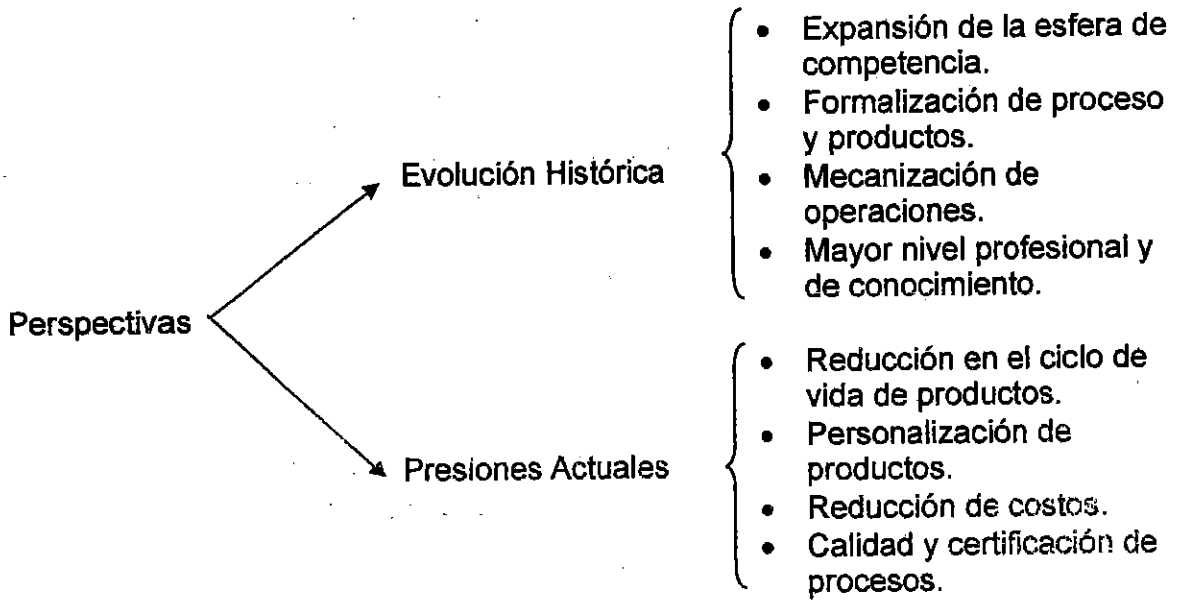
INGENIERÍA DE SOFTWARE

Definiciones

- **IEEE -** El uso de métodos sistemáticos, disciplinados y cuantificables para el desarrollo, operación y mantenimiento de software. (Es decir, la aplicación de prácticas de Ingeniería de Software). O bien, el estudio de técnicas relacionadas con lo anterior.
- **Fairley -** Disciplina tecnológica y de administración que se ocupa de la producción y evolución sistemática de productos de software que son desarrollados y modificados dentro de los tiempos y costos estimados.
- **Ghezzi -** Campo de la ciencia de la computación que trata con la construcción de sistemas de software que son tan grandes o complejos que son construidos por un equipo o equipos de ingenieros.

Conocimientos requeridos

- Principios teóricos de representación y computación.
- Aplicación de métodos formales.
- Uso de notaciones de modelización, especificación, diseño y programación.
- Combinación de conocimientos de:
  - ✓ Metodologías.
  - ✓ Tecnologías.
  - ✓ Técnicas de administración de proyectos


Perspectivas
Evolución Histórica
Presiones Actuales
• Expansión de la esfera de competencia.
• Formalización de proceso y productos.
• Mecanización de operaciones.
• Mayor nivel profesional y de conocimiento.
• Reducción en el ciclo de vida de productos.
• Personalización de productos.
• Reducción de costos.
• Calidad y certificación de procesos.

Definida qué es la ingeniería de Software, se considera adecuado definir:

> **REINGENIERÍA DE SOFTWARE**: Se puede definir como la modificación de un producto de software, o de ciertos componentes, usando para el análisis del sistema existente técnicas de ingeniería inversa y, para la etapa de reconstrucción, herramientas de ingeniería directa, de manera tal que se oriente este cambio hacia mayores niveles de facilidad de mantenimiento, reutilización, comprensión y evaluación".

Cuando una aplicación lleva siendo usada años, puede volverse inestable como fruto de múltiples correcciones, adaptaciones o mejoras a lo largo del tiempo. Esto deriva en que cada vez que se pretenda realizar un cambio se producen efectos laterales inesperados, y hasta graves, por lo que, si se pretende que la aplicación siga siendo de utilidad, se debe aplicar reingeniería.

Entre los beneficios de aplicar reingeniería a un producto existente se puede incluir:

- Reducir los riesgos evolutivos de una organización
- Ayudar a las organizaciones a recuperar sus inversiones en software
- Mejorar el mantenimiento del software
- Ampliar las capacidades de las herramientas CASE

La reingeniería involucra actividades diferentes tales como:

- **Análisis de inventarios**: Del inventario de aplicaciones de una organización, son candidatos a la reingeniería aquellas aplicaciones más importantes para la organización, las más longevas y las menos mantenibles en la actualidad.

- **Reestructuración de documentos**: En general se utiliza un enfoque de "documentar cuando se toque" debido a los altos costos de mantener la documentación de software actualizada.

- **Ingeniería inversa**: Consiste en reconstruir información de especificación de software a partir del producto de software funcionando. Es el proceso de análisis de un programa con el fin de crear una representación del mismo con un nivel de abstracción mayor que el código fuente.

- **Reestructuración de código**: Requiere analizar el código fuente empleando una herramienta de reestructuración, indicando las violaciones a las estructuras de programación estructurada, reestructurando el código (puede ser automáticamente). El código reestructurado se revisa y comprueba para asegurar que no se hayan introducido anomalías. Se actualiza la documentación interna.

- **Reestructuración de datos**: Se analiza minuciosamente la arquitectura de datos actual y se definen los modelos de datos necesarios, se identifican los objetivos de los datos y los atributos, y se revisa l calidad de las estructuras de datos existentes. Cuando la estructura de datos es débil, se aplica la reingeniería de datos.

- **Ingeniería directa**: No sólo recupera la información de diseño a partir del software existente, sino que también utiliza esta información para alterar o reconstruir el sistema con la finalidad de mejorar su calidad global.

En la mayoría de los casos, el software sometido a reingeniería vuelve a implementar la función del sistema y también añade funciones y mejoras.

14. La importancia del Software
14.1 El Software y el Proceso

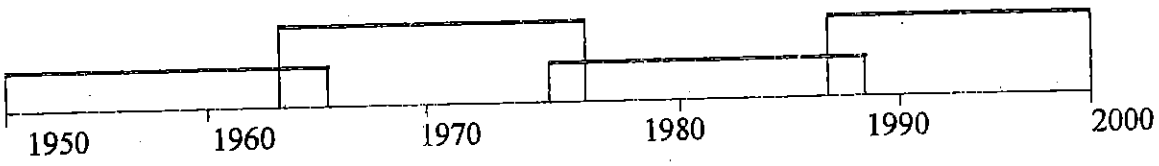
Durante las tres primeras décadas de la Informática, el principal desafío era el desarrollo del Hardware de computadoras de forma de reducir el costo de procesamiento y almacenamiento de datos.

A partir de la década del 80, cuando toda la potencia de cálculo y almacenamiento de las grandes computadoras está disponible en una PC, y dado que el Software es el mecanismo que nos facilita usar y explotar ese potencial, el principal desafío se ha transformado en el de mejorar la calidad (y reducir el costo) de las soluciones basadas en computadoras, soluciones que se implementan con el Software.

14.2 La evolución del software

La siguiente figura describe la evolución del software en el contexto de las áreas de aplicación de los sistemas basados en computadoras:

Los primeros años	La segunda era	La tercera era	La cuarta era
• Orientación por lotes • Distribución limitada. • Software a medida	• Multiusuarios. • Tiempo real. • Bases de datos. • Software como producto	• Sistemas distribuidos. • Incorporación de la inteligencia. • Hardware de bajo costo. • Impacto en el consumo	• Potentes sistemas de sobremesa. • Tecnología OO. • Sistemas expertos. • Redes neuronales artificiales. • Computación paralela.



14.3 El software

Se podría describir el software de la siguiente manera:

SOFTWARE:

Instrucciones que cuando se ejecutan proporcionan la función y el comportamiento deseado.

Estructuras de datos que facilitan a los programas manipular adecuadamente la información.

Documentos que describen la operación y uso de los programas.

14.4 Características del Software

a. El software se desarrolla, no se fabrica en el sentido clásico.
b. El software no se "estropea' ya que no es susceptible a los males del entorno.
c. La mayoría del software se construye a medida, en lugar de ensamblar componentes existentes.

14.5 Componentes del Software

Las componentes del software se crean mediante una serie de traducciones que hacen corresponder los requisitos del cliente con un código ejecutable en una máquina.

Las traducciones que se llevan a cabo son:

modelo de requisitos (prototipo) →diseño

diseño del software → lenguaje de programación que especifique estructuras de datos, atributos, procedimientos y requisitos que atañen al software

forma en lenguaje → instrucciones ejecutables en la máquina de programación

La **reusabilidad** es una característica muy importante para una componente de software de alta calidad. Es decir, una componente debe diseñarse e implementarse para que pueda volver a usarse en muchos programas diferentes.

Esta reusabilidad se extiende actualmente, no sólo a programas o bibliotecas de subrutinas, es decir algoritmos, sino hasta las estructuras de datos.

Una componente reusable, en ésta década, **encapsula** tanto datos como procesos en un único paquete (llamado **clase** u **objeto**) permitiendo al Ingeniero de Software crear nuevas aplicaciones a partir de software reusable.

14.6 Aplicaciones del Software

El software puede aplicarse en cualquier situación en la que se haya definido previamente un conjunto específico de pasos procedimentales (es decir, un algoritmo).

Veamos las siguientes áreas del software que indican la amplitud de las áreas de aplicación:

Software de Sistemas Conjunto de programas escritos para servir a otros programas. Implica una fuerte interacción con el Hardware, una gran utilización de múltiples usuarios, operación concurrente que requiere planificación, compartición de recursos, y gestión de procesos, estructuras de datos complejas y múltiples interfaces externas: compiladores, editores, utilitarios de gestión de archivos, componentes de los Sistemas Operativos, utilitarios de manejo de periféricos, procesadores de telecomunicaciones, etc.

Software de tiempo real Es el software que mide/analiza y controla sucesos del mundo real. Se incluyen: una componente de adquisición de datos que recolecta y da formato a la información recibida del entorno externo, una componente de análisis que transforma la información según lo requiera la aplicación, un componente de control/salida que responda al entorno externo y una componente de monitorización que coordine los demás componentes, de manera tal que pueda mantenerse la respuesta en tiempo real (rango: 1 milisegundo a 1 minuto).

Software de Gestión El procesamiento de información comercial constituye la mayor de las áreas de aplicación del Software. Los sistemas "discretos" (nóminas, cuentas de haberes, inventarios, etc.) han evolucionado hacia el software de Información de Gestión (SIG) que accede a una o más grandes bases de datos que contienen información comercial.

Software de Ingeniería y Científico Se caracteriza por lo algoritmos de manejo de números. Se usa en muchos campos de la ciencia: astronomía hasta vulcanología, análisis de la presión de los automotores, dinámica orbital, biología molecular, etc.

Software empotrado Es el software que reside en ROM y se usa para controlar productos y sistemas de los mercados industriales y de consumo. Es software inteligente que puede ejecutar funciones muy limitadas y curiosas (ej.: El control de las

teclas de un horno microondas) con funciones con capacidad de control (en un automóvil control de gasolina, sistemas de frenado, etc.)

Software de P.C. Son innumerables las aplicaciones: procesadores de textos, planillas de cálculos, gráficos, entretenimientos, gestión de bases de datos, aplicaciones financieras, de negocios y personales, redes, etc.

Software de Inteligencia Artificial Usa algoritmos no numéricos para resolver problemas complejos para los que no son adecuados el cálculo o el análisis directo. Su área más activa son los sistemas expertos. Otra área de aplicación del software de I.A. es el reconocimiento de patrones (imágenes y voz), la prueba de teoremas y juegos. En los últimos años han surgido las llamadas redes neuronales artificiales que simulan la estructura de proceso del cerebro y a la larga pueden llevar a una clase de software que reconozca patrones complejos y aprenda de "experiencia" pasada.

14.7Software: Una crisis en el horizonte

El término alude a un conjunto de problemas que surgen en el desarrollo de software, tales como: ¿Cómo desarrollar software?.¿Cómo mantener el volumen cada vez mayor de software existente?.¿Cómo ~~mantenemos~~ mantenernos al corriente de la demanda creciente de software?.

Básicamente, **los problemas** que afligen al desarrollo de software se centran en ciertos aspectos "de fondo":

a) La planificación y estimación de costos son frecuentemente imprecisas.

b) La "productividad" de la comunidad del software no se corresponde con la demanda de servicios.

c) La calidad del software no llega, a veces, a ser ni aceptable.

Los problemas de este tipo producen insatisfacción y falta de confianza en el cliente y son las manifestaciones visibles de otras dificultades del software, tales como: no se recogieron datos sobre el proceso de desarrollo del software, las comunicaciones entre el cliente y el desarrollador fue escasa, la calidad del software es cuestionable, el software puede ser difícil de mantener, etc.

Todos estos problemas se pueden corregir dándole un enfoque de Ingeniería al desarrollo de software.

14.8Paradigmas de la Ingeniería de Software

Mediante la combinación de métodos completos para todas las fases de desarrollo de software, mejores herramientas para automatizar estos métodos, bloques de construcción más potentes para la implementación de software, mejores técnicas para la garantía de calidad del software y una filosofía predominante para la coordinación, control y gestión podemos conseguir una disciplina para el desarrollo de software: la **Ingeniería de Software.**

Fritz Bauer, en 1969, propone la siguiente definición de Ingeniería de Software:

> **El establecimiento y uso de principios de Ingeniería robustos, orientados a obtener software económico que sea fiable y funcione de manera eficiente sobre máquinas reales.**

La ingeniería de software surge de la ingeniería de sistemas y de hardware, abarca tres elementos claves: **Métodos, Herramientas y Procedimientos** que facilitan al gestor controlar el desarrollo del software y suministrar las bases para construir software de alta calidad en forma productiva.

Métodos: Indican como construir técnicamente el software. Entre las tareas que abarcan se incluye la planificación y estimación de proyectos, análisis de requisitos de sistema y del software, diseño de estructuras de datos, arquitectura de programas y procedimientos algorítmicos, codificación, prueba y mantenimiento. Generalmente, cada método introduce una flotación especial, gráfica u orientada a un lenguaje y un conjunto de criterios para la calidad del software.

Herramientas: Suministran un soporte automático o semiautomático para los métodos. CASE combina software, hardware y bases de datos sobre Ingeniería de Software.

Procedimientos: Son el "pegamento" que une los métodos y las herramientas y facilita un desarrollo racional y oportuno del software. Definen la secuencia en la que se aplican los métodos, las entregar que se requieren, los controles que ayudan a asegurar la calidad y coordinar los cambios y las directrices que ayudan a los gestores del software a evaluar el proceso.

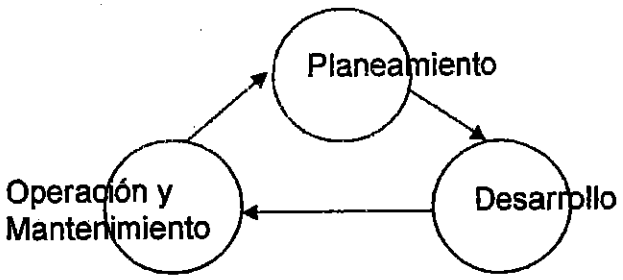
La Ingeniería de Software está compuesta por una serie de pasos, frecuentemente llamados **PARADIGMAS** de la Ingeniería de Software de los cuales discutiremos algunos.

Ciclo de vida de desarrollo de software

El **proceso de desarrollo de software** es un conjunto de actividades que generan productos intermedios, lo que, a su vez son transformados en otros productos por otros procesos. Esta es la visión del **ciclo de vida**: una cadena de actividades con productos intermedios.

Esta visión evoca la visión de los organismos vivos que tienen un nacimiento, una vida y una madurez y muerte. También se puede ver como una secuencia de estados.

Ciclo de vida de sistemas



El desarrollo de sistemas software se hace usando un modelo de proceso.

Proceso

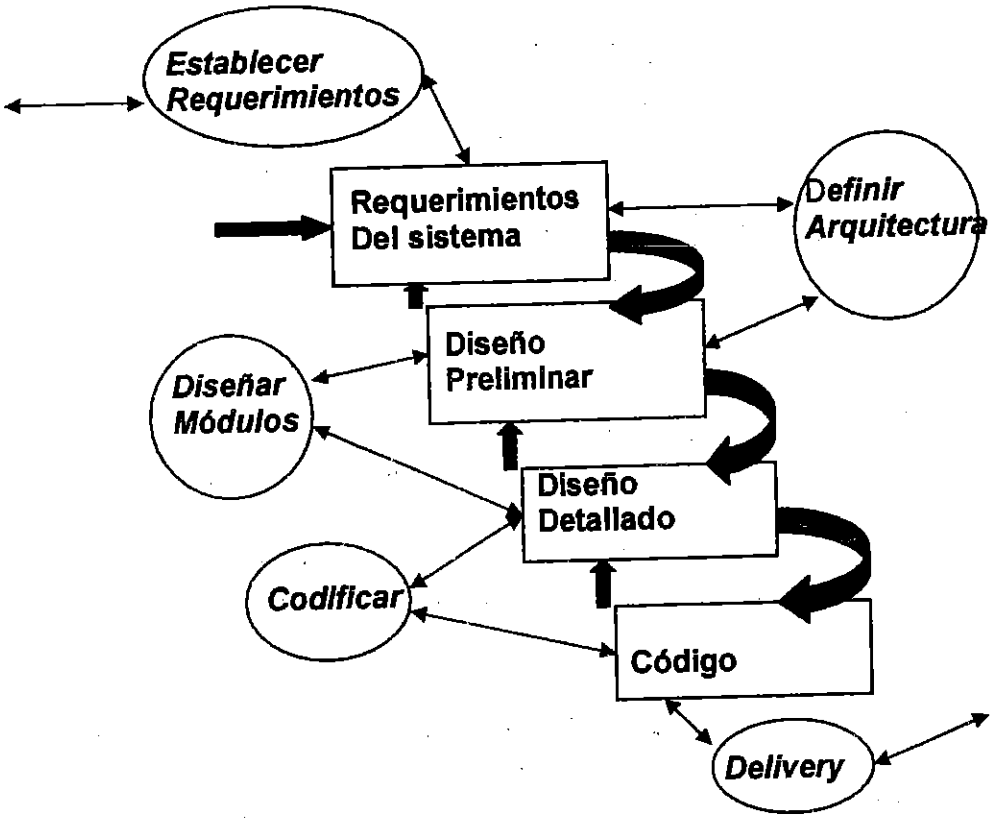
Es un conjunto de pasos que involucran actividades, restricciones y recursos que producen un output de algún tipo. Involucra **técnicas**, **herramientas** y **procedimientos**.

> **Un proceso de software es un conjunto de actividades, métodos, prácticas y transformaciones que la gente usa para desarrollar y mantener software y los productos asociados.**

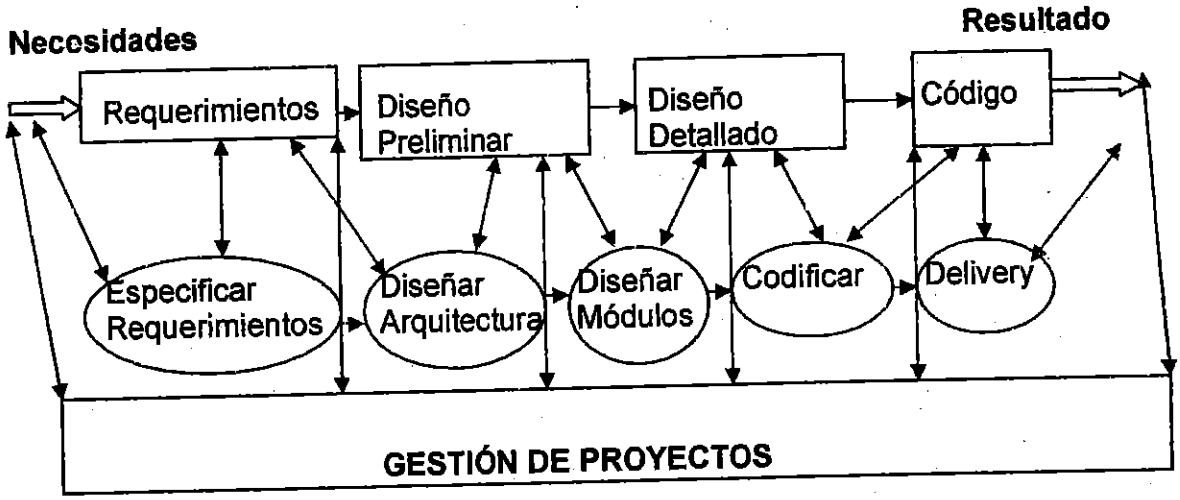
Definiciones:

. **Técnica**:	Conjunto de pasos a seguir para producir algún resultado. (Ej. Lectura de código)
. **Herramienta**	Instrumento o sistema automatizado para alcanzar algo en una mejor forma (Ej. Analizador de código)
. **Método/Procedimiento**	Combinación de herramientas y técnicas que en conjunto producen un producto particular (Ej. Inspección de código).

Procesos del desarrollo de software



Modelo Completo



El **modelo abstracto** se obtiene:

- Visualizando cada rectángulo superior como un **estado** diferente.
- Visualizando cada círculo como una **actividad** diferente.

Los dos conceptos:

- **Ciclo de vida**: Visión del proceso de software que considera la presencia de productos.
- **Ciclo de vida del software**: Conjunto de métodos que cubren el ciclo de vida completo: modelo de desarrollo incremental, uso de diseño estructurado.
  - Modelado de ciclo de vida
  - Modelo de proceso de software

se pueden considerar como sinónimos a los efectos de la preocupación en el proceso.

Modelos de procesos de software

- El modelo prescribe todas las actividades principales del proceso.
- El proceso usa recursos y produce productos intermedios y finales.
- El proceso puede estar formado por sub-procesos.
- Cada actividad de proceso tiene un criterio de entrada y salida.
- Las actividades son organizadas en una secuencia.
- Cada proceso tiene un conjunto de principio de guía.
- Se pueden aplicar restricciones o controles a una actividad, a un recurso o a un producto.

El uso de modelos de procesos:

a) Ayuda a una comprensión común
b) Ayuda a encontrar inconsistencias, redundancias y omisiones
c) El modelo debería reflejar los objetivos de desarrollo
d) Permite evaluar actividades candidatas para atacar estos objetivos
e) Permite la adecuación a cada situación particular.

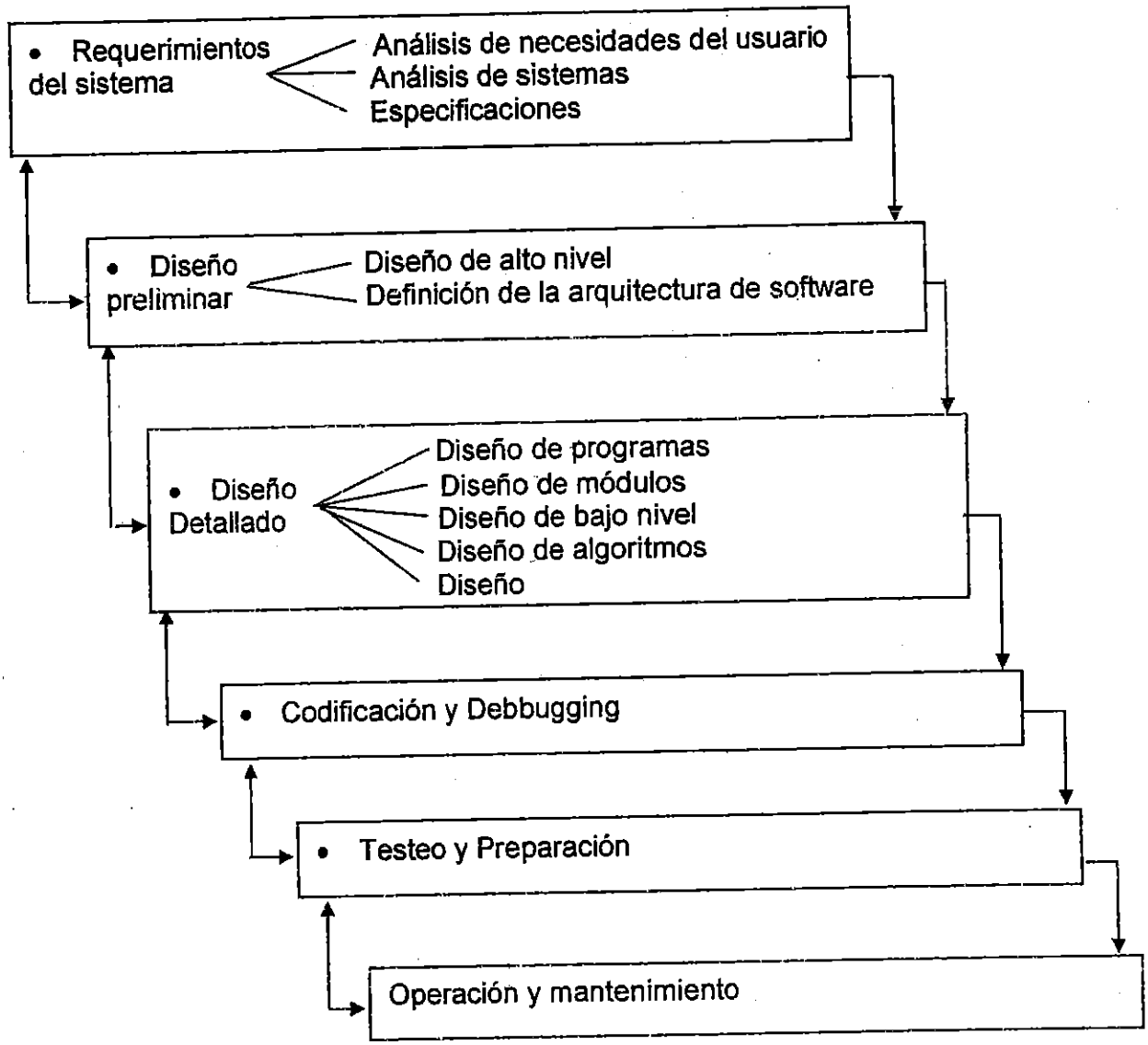
Modelos y Métricas

Las mediciones capturan información sobre el proceso que se está ejecutando.

Permiten:

- Evaluar el proceso
- profundizar el conocimiento del producto
- detectar debilidades del ambiente
- proveer conocimiento sobre la mejora del proceso
- ayudar a la evolución del proceso

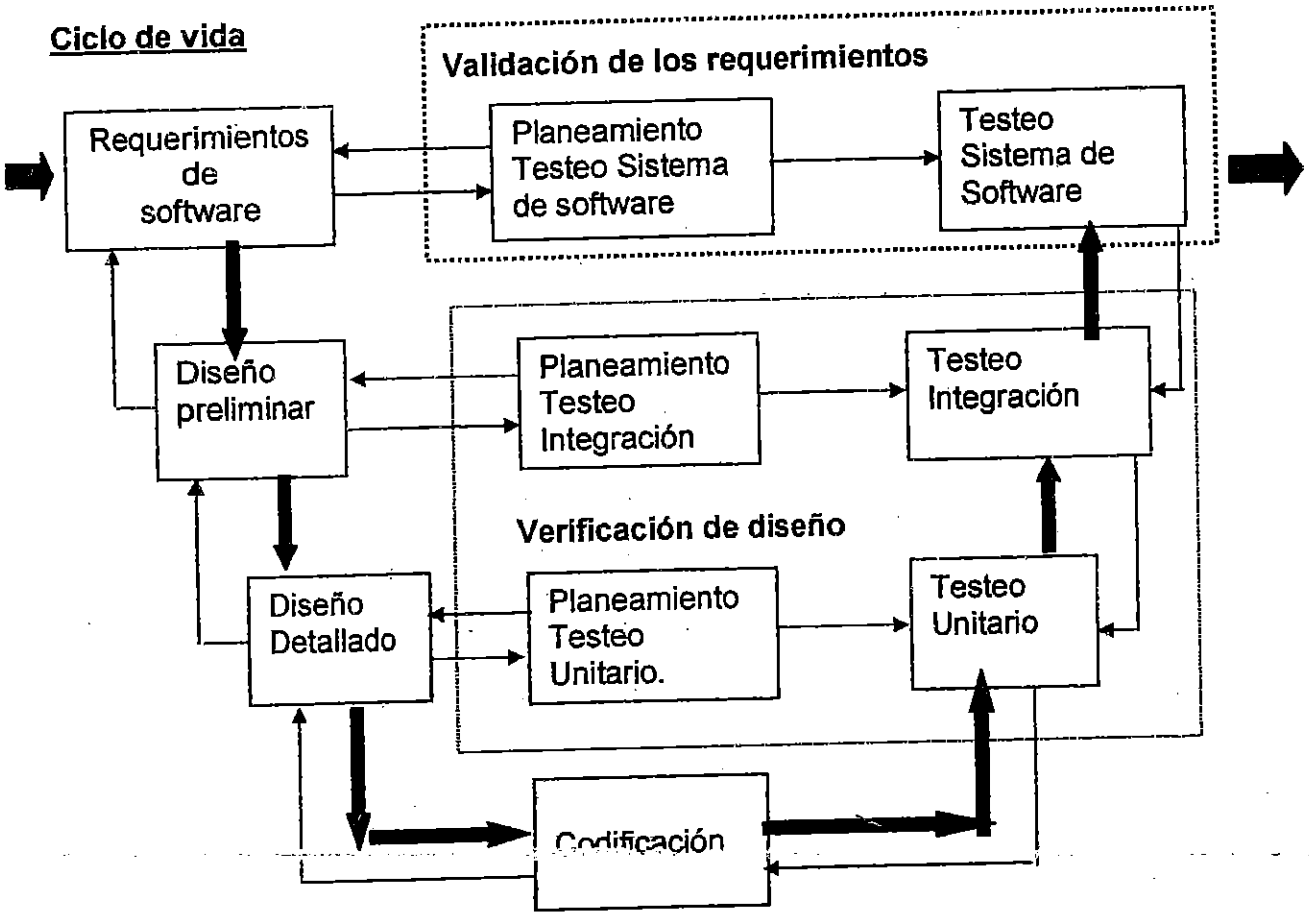
Modelo en cascada



Modelo de cascada clásico

- Requiere definir qué hará el sistema antes de construirlo (Requerimientos/Diseño).
- Requiere plantear la interacción de las partes antes de construirlas (Diseño/codificación).
- Los gerentes deben seguir muy de cerca el proyecto y corregir muy tempranamente.
- Requiere producir una serie de documentos que luego se usan en el testeo y mantenimiento.
- Reduce los costos de desarrollo y mantenimiento como consecuencia de las razones anteriores.
- Permite a la organización de desarrollo ser más estructurada y organizada.

Ciclo de vida



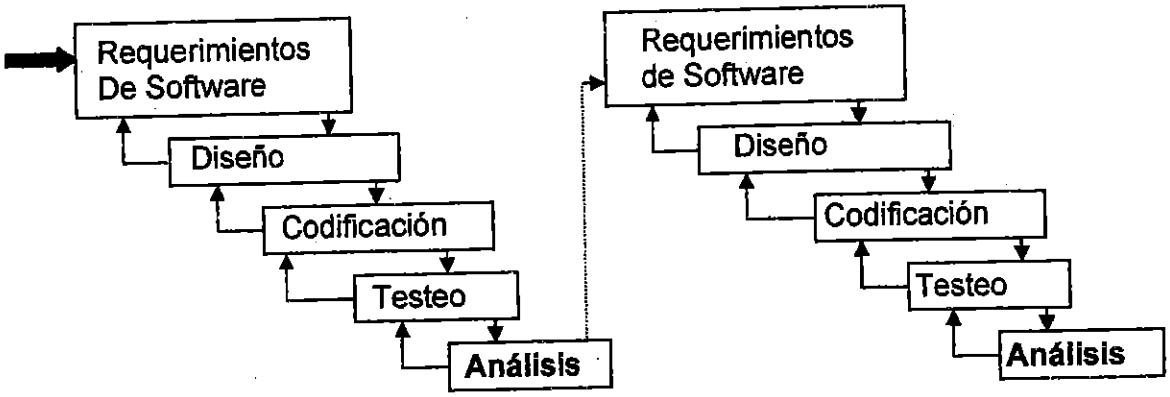
Problemas del modelo de cascada

- El desarrollo de software es iterativo en su esencia.
- ¿Cómo se transforman los productos?
- El desarrollo es una actividad de resolución de problemas.
- En los hechos, de una etapa se pasa a cualquiera de las otras.
- Se ha convertido en el modelo "ideal" típico.

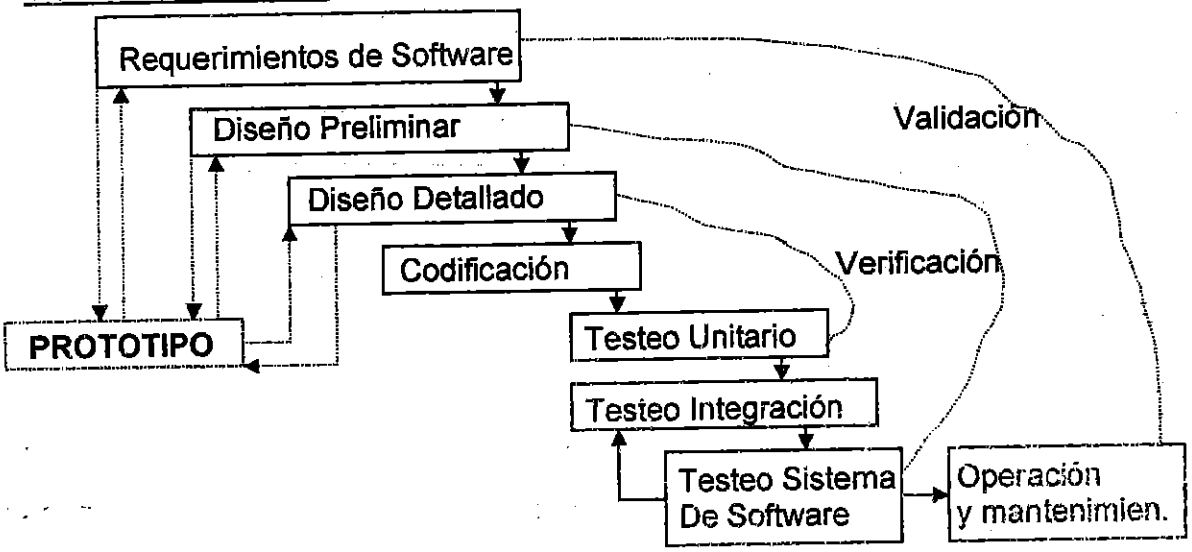
Prototipos: Tipología

- **Escenarios o simulaciones**: Herramientas para entender o validar los requerimientos del usuario.
- **Rápidos desechables**: Construcción "quick and dirty" que ataca un aspecto particular, para que el usuario potencial lo use durante un tiempo y provea información al equipo.
- **Evolutivos**: Atacan una funcionalidad acotada del sistema que resuelve algunos de los requerimientos del usuario.

Mejora Iterativa



Modelo de Prototipo



Entrega en etapas

Procesos del ciclo de vida del software

Se clasifican en:

- Proceso del ciclo de vida primario. --- Conducen las principales funciones.
- Procesos del ciclo de vida de soporte --- Dan soporte a otros procesos para realizar una función especial.
- Procesos del ciclo de vida organizacional --- Establecen, controlan y mejoran los procesos del ciclo de vida.

A su vez, cada una de estas categorías se clasifica en:

- Procesos Primarios
  - Proceso de adquisición — Tareas del que contractualmente adquiere un producto de software o un servicio.
    Actividades: Definiciones de la necesidad, pedido de propuesta (RFP), selección del proceso hasta la aceptación del sistema.
  - Proceso de provisión — El servicio puede ser el desarrollo de un producto, un sistema conteniendo software, la operación o mantenimiento de un sistema de software.
    Actividades: Preparar una propuesta a un adquiridor, acordar un contrato, identificar los procedimientos y recursos necesarios, desarrollar, ofrecer el servicio.
  - Proceso de desarrollo — Actividades y tareas del desarrollador del sistema y del software.
    Desarrollar software se aplica a software nuevo o a modificar el ya existente.
    Actividades: tareas propias del desarrollo del software.
  - Proceso de operación — Abarca la operación del software y el soporte operacional a los usuarios.
    Actividades: implementación, testeo operativo, operación del sistema y soporte del usuario.
  - Proceso de mantenimiento — Causa: error, necesidad de mejora, adaptación.
    Alcance: modificar el código y la documentación asociada, finaliza cuando el sistema se retira de uso.
    Actividades: análisis del problema y modificación, aceptación, migración, retiro del software.

- Procesos de soporte
  - Proceso de documentación: Su objetivo es el registro de la información producida por los procesos del ciclo de vida.
  - Proceso de gestión de configuración: Identifica y define el baseline de los ítems de software en el sistema, para controlar modificaciones y versiones de los ítems.
  - Proceso de aseguramiento de la calidad: Provee un framework para asegurar (el adquirente o el cliente) el cumplimiento de los productos o servicios con sus requerimientos contractuales y el acotamiento de los planes establecidos.
  - Proceso de verificación: Determina si los requerimientos para un sistema son completos y correctos y que el output de una actividad satisface los requerimientos y condiciones impuestos en actividades previas.
  - Proceso de validación: Determina si el sistema final cumple con el objetivo inicial. No reemplaza otras evaluaciones, sólo los complementa.
  - Proceso de revisión conjunta: Provee un framework para la interacción entre el "revisado" y el revisor.
  - Proceso de auditoría: El auditor evalúa los productos y actividades del auditado con énfasis en el cumplimiento de los requerimientos y planes.
  - Proceso de resolución de problemas: Se resuelven los problemas tomando acciones correctivas en la medida que se detecten.

- Procesos organizacionales
  - Proceso gerencial: Define actividades y tareas del manager.
  - Proceso de infraestructura: La infraestructura puede incluir hardware, software, standards, herramientas, técnicas y medios. Objetivo: definir las actividades para establecer y mantener la infraestructura necesaria.
  - Proceso de mejora: Objetivos: mejorar el proceso organizacional. Actividades: evaluar, medir, controlar y mejorar el proceso de ciclo de vida.
  - Proceso de entrenamiento: Actividades: identificar y hacer un plan para incorporar o desarrollar recursos de personal.

Comparación de modelos

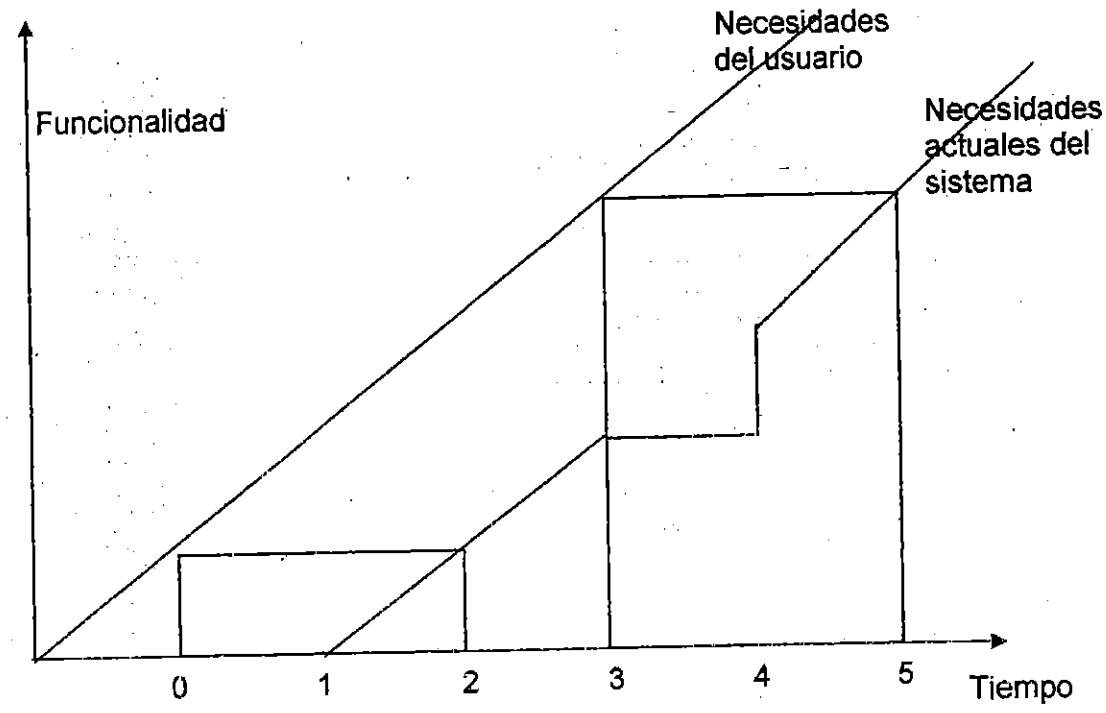
Los nuevos modelos son:

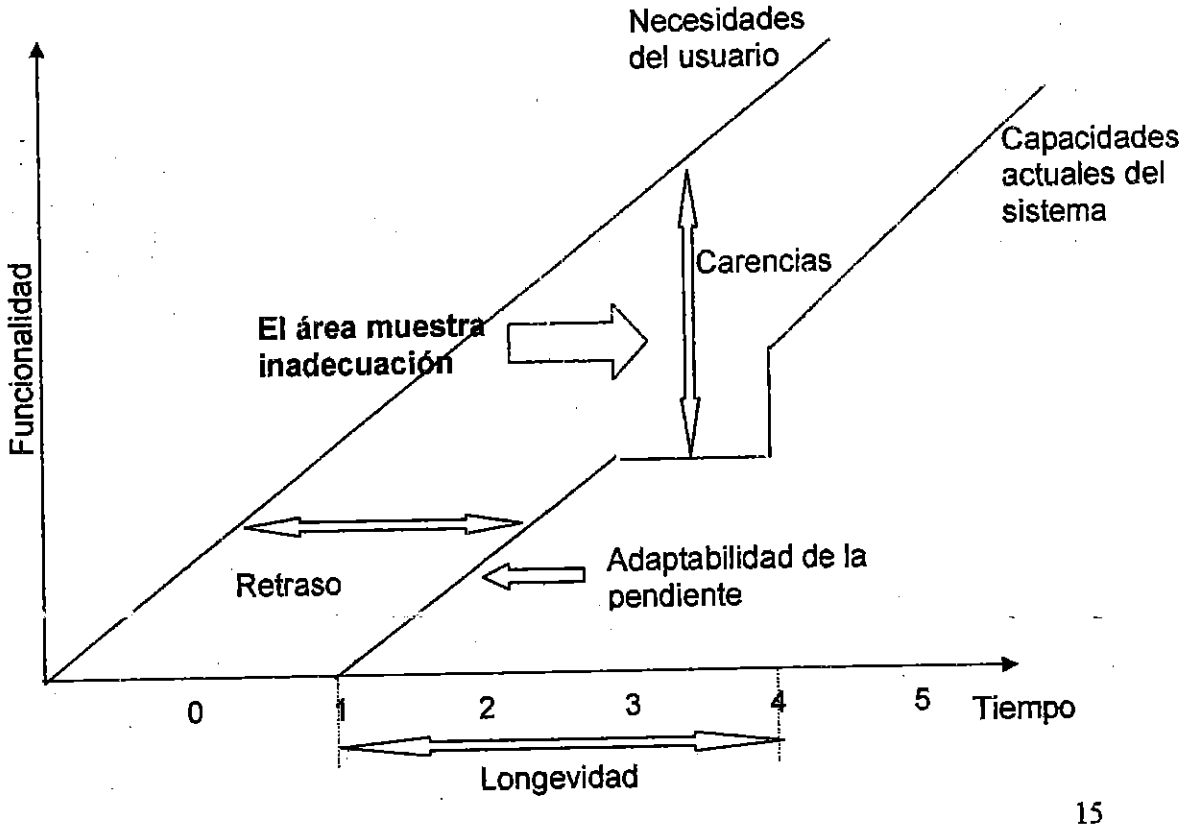
- Prototipo rápido desechable
- Desarrollo incremental
- Prototipo evolutivo
- Software reusable
- Síntesis automática de software

Las necesidades del usuario se expresan por la funcionalidad del sistema como función monótona creciente del tiempo.

Se utilizan varias métricas:

- Carencias: distancia sistema-necesidades en t
- Retraso: tiempo para satisfacer una necesidad.
- Adaptabilidad: ajuste a nuevos requerimientos.
- Longevidad: tiempo que una solución es viable.
- Inadecuación: carencias en un plazo de tiempo


Necesidades
del usuario
Funcionalidad
Necesidades
actuales del
sistema
0
1
2
3
4
5
Tiempo


Necesidades
del usuario
Capacidades
actuales del
sistema
Carencias
Funcionalidad
El área muestra
inadecuación
Retraso
Adaptabilidad de la
pendiente
0
1
2
3
4
5
Tiempo
Longevidad

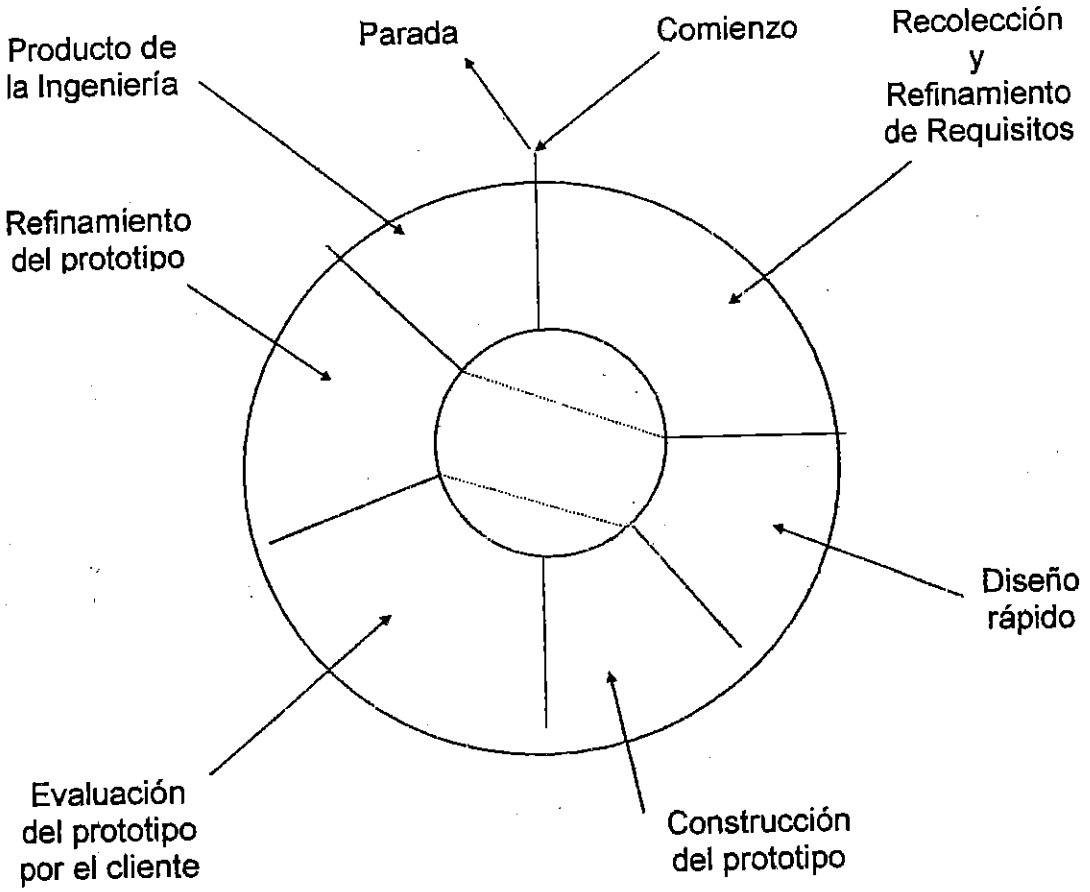
14.8.2 La construcción de prototipos

La construcción de prototipos es un proceso de la Ingeniería de Software que facilita al programador la **creación de un modelo** del software a construir. El modelo tomará una de las tres formas siguientes:

i) **Un prototipo en papel** o un modelo basado en PC que describa la interacción hombre-máquina de forma que facilite al usuario la comprensión de como será la interacción.

ji) Un prototipo que implemente algunos **subconjuntos de la función** requerida del programa deseado.

iii) Un **programa existente** que ejecute parte o toda la función deseada pero que tenga otras características que deban ser mejoradas.



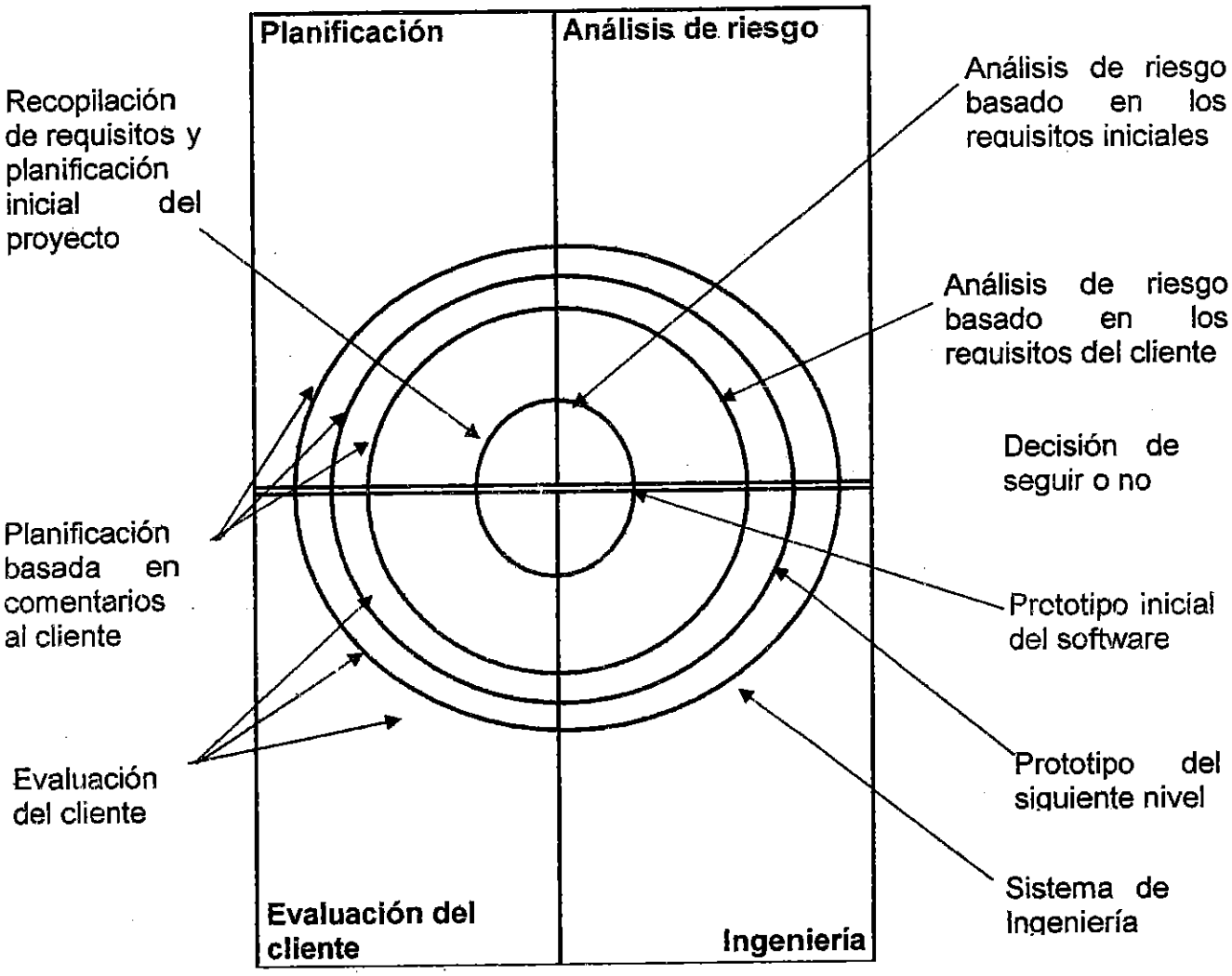
Idealmente, **el prototipo sirve como mecanismo para identificar los requisitos del software.**

La construcción de prototipos es un paradigma efectivo de la Ingeniería de Software, si se definen al comienzo las reglas del juego, es decir, que el cliente y el técnico estén de acuerdo en que el prototipo se construya para servir sólo como mecanismo de definición de requisitos. Es frecuente que ambos caigan en la tentación de mejorarlo o arreglarlo para que sea la base del sistema. Una vez definidos los requisitos, el prototipo debe ser descartado para construir un software real con los ojos puestos en la calidad y el mantenimiento.

14.8.3 El modelo en espiral

Este modelo cubre las mejores características del ciclo de vida clásico y la creación de prototipos y añade a éstos un análisis de riesgos que no había sido considerado.

Define **cuatro actividades principales,** representadas por los la siguiente figura:



En otras palabras, las cuatro actividades de las que consta el modelo son:

1. **Planificación**: determinación de objetivos, alternativas y restricciones.
2. **Análisis de riesgo:** análisis de alternativas e identificación / resolución de riesgos.
3. **Ingeniería**: desarrollo del producto de "siguiente" nivel.
4. **Evaluación del cliente**: valoración de los resultados de la ingeniería.

En cada vuelta del espiral se construyen sucesivas versiones del software, cada vez más completas. Si el análisis de riesgo indica que hay incertidumbre en los requisitos, se puede usar el cuadrante de ingeniería para dar asistencia tanto al encargado del desarrollo como al cliente. El cliente evalúa el trabajo de ingeniería y sugiere modificaciones.

En cada vuelta del espiral, la culminación del análisis de riesgos resulta en una **decisión de seguir o no seguir.**

Este paradigma es actualmente el **enfoque más realista** en el desarrollo de software y de sistemas a gran escala. Usa un enfoque **"evolutivo"** permitiendo al desarrollador y al cliente entender y reaccionar a los riesgos en cada nivel evolutivo.

Pero aún así, requiere una considerable habilidad para la valoración del riesgo, y cuenta con esa habilidad para el éxito. Es un modelo relativamente nuevo que no ha sido tan usado como los precedentes.

14.8.4 Técnicas de cuarta generación

Este término abarca un **amplio espectro de herramientas de software** que tienen algo en común: Todas facilitan, al que desarrolla el software, la especificación de algunas características del software de alto nivel. Luego, la herramienta genera automáticamente el código fuente basándose en la especificación del técnico. Es cada vez más evidente que:

> **Cuanto mayor sea el nivel en el que se especifique el software, más rápido se podrá construir el programa.**

Un ejemplo de estas técnicas es el paradigma TG4 que se orienta hacia la posibilidad de especificar el software a un nivel más próximo al lenguaje natural o a una notación que proporcione funciones significativas. Pero las herramientas actuales no son lo suficientemente sofisticadas para entender el lenguaje natural, y no lo serán por algún tiempo.

14.8.5 Combinación de paradigmas

En muchos casos los paradigmas pueden y deben combinarse de forma de poder **utilizar las ventajas** de cada uno e un único proyecto.

No hay necesidad de ser dogmático en la elección de paradigmas para la Ingeniería de Software; la naturaleza de la aplicación debe dictar el método a elegir. Mediante la combinación de paradigmas, el todo puede ser mejor que la suma de las partes.

14.9Una visión genérica de la Ingeniería de Software

El proceso de Ingeniería de Software contiene **tres fases genéricas,** independientes del paradigma de ingeniería elegido, del área de aplicación, del tamaño del proyecto o de su complejidad: **definición, desarrollo y mantenimiento.**

La fase de definición se centra en el qué. Es decir, intenta identificar qué información ha de ser procesada, qué función y rendimiento se desea, qué interfaces han de establecerse, qué restricciones de diseño existen, y qué criterios de validación se necesitan para definir un sistema correcto.

Los **pasos específicos involucrados** en esta fase son:

- **Análisis del sistema:** define el papel de cada elemento de un sistema informático, asignando finalmente al software el papel que va a desempeñar.

- **Planificación del proyecto de software**: Una vez establecido el ámbito del software, se analizan los riesgos, se asignan los recursos, se estiman los costos, se definen las tareas y se planifica el trabajo.

- **Análisis de requisitos:** Antes de continuar, es necesario disponer de una información más detallada del ámbito de información y de función del software.

La fase de desarrollo se centra en el como. Intenta descubrir como han de diseñarse las estructuras de datos y la arquitectura del software, como han de implementarse los detalles procedimentales, como ha de traducirse el diseño a un lenguaje de programación y como ha de realizarse la prueba.

Los **pasos que implica esta fase** son:

- **Diseño del Software:** Traduce los requisitos del software a un conjunto de representaciones que describen la estructura de los datos, la arquitectura, el procedimiento algorítmico y las características de la interfaz.

- **Codificación:** Donde las representaciones del diseño son traducidas a un lenguaje artificial (de programación o no procedimental) dando como resultado instrucciones ejecutables en una computadora.

- **Prueba del Software**: Una vez que el software ha sido implementado en una forma ejecutable por la máquina, debe ser probado para descubrir los defectos que puedan existir en la función, la lógica y en las implementaciones.

La fase de mantenimiento se centra en el cambio. Este cambio va asociado a la corrección de errores, a las adaptaciones requeridas por la evolución del entorno del software y a las modificaciones requeridas debido a los cambios de los requisitos del cliente dirigidos a reforzar o ampliar el sistema.

Durante esta fase se presentan **tres tipos de cambios:**

- **Corrección**: El mantenimiento correctivo cambia el software para corregir los defectos u errores detectados a llevar a cabo actividades de garantía de calidad.

- **Adaptación**: El mantenimiento adaptativo consiste en modificar el software para acomodarlo a los cambios del entorno externo (La UCP, el sistema operativo, los periféricos, etc.)

- **Mejora** : El mantenimiento perfectivo amplía el software más allá de sus requisitos funcionales originales.

ARQUITECTURA DE SOFTWARE

La arquitectura de un programa o sistema de software, es la estructura de ese sistema, que incluye componentes de software, las propiedades visibles externas de esos componentes y las relaciones entre éstos. El término también puede incluir la documentación sobre la arquitectura de software del sistema.

> **Una arquitectura de software consiste en un conjunto de patrones y abstracciones coherentes que proporcionan el marco de referencia necesario para guiar la construcción del software para un sistema de información.**

Esta disciplina de la ingeniería de software comienza a estudiarse a principios de los años 90 aunque los ingenieros de software hayan empleado arquitecturas de software desde siempre, casi sin darse cuenta.

La arquitectura es un conjunto de decisiones principales de diseño acerca de un sistema de software sobra la que deben tenerse claros los siguientes principios:

- Cada aplicación tiene una arquitectura
- Cada aplicación tiene al menos un arquitecto
- La arquitectura no es una etapa del desarrollo.

El contexto de trabajo para definir la arquitectura de un sistema de software es:

- Los requerimientos
- El diseño
- La implementación del Software
- El análisis y testeo
- La evolución
- Los procesos de desarrollo

Algunos objetivos dentro de un esquema de Arquitectura de software pueden ser: el software debe ser mantenible (fácilmente analizable, corregible y modificable); el nivel de interacción con otros sistemas o su escalibilidad.

Ejemplos de arquitecturas:

- Monolíticas : Los grupos funcionales del software están altamente acoplados entre sí.

•cliente-servidor: se reparte la carga de cómputo en dos partes independientes.

•Tres niveles: La carga se divide en tres partes; presentación, cálculo y almacenamiento.

Hoy en día, la disciplina estudia enfáticamente el uso de patrones de arquitectura.

INGENIERÍA DE REQUERIMIENTOS

Se puede decir que Ingeniería de requerimientos es el **proceso de establecer los servicios que el cliente requiere del sistema y los límites bajo los cuales se desarrolla y opera.**

Los requerimientos pueden ser:

- Funcionales: describen servicios o funciones
- No-funcionales: son un límite en el sistema o proceso de desarrollo.

La ingeniería de requerimientos se compone de:

- **Definición de requerimientos**: Una declaración en un lenguaje natural incluye los diagramas de los servicios del sistema y sus límites operacionales. Es escrito por los clientes.
- **Especificación de requerimientos**: Un documento estructurado con descripción y detalle de los servicios del sistema. Escrito como un contrato entre el cliente y el contratista.
- **Especificación de software**: Descripción detallada del software, la cual debe servir como base para diseño e implementaciones. Escrito por desarrolladores.

Métodologías Ágiles en el Desarrollo de Software

José H. Canós, Patricio Letelier y Mª Carmen Penadés

DSIC -Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia
{ jhcanos | letelier | mpenades }@dsic.upv.es

RESUMEN

El desarrollo de software no es una tarea fácil. Prueba de ello es que existen numerosas propuestas metodológicas que inciden en distintas dimensiones del proceso de desarrollo. Por una parte tenemos aquellas propuestas más tradicionales que se centran especialmente en el control del proceso, estableciendo rigurosamente las actividades involucradas, los artefactos que se deben producir, y las herramientas y notaciones que se usarán. Estas propuestas han demostrado ser efectivas y necesarias en un gran número de proyectos, pero también han presentado problemas en otros muchos. Una posible mejora es incluir en los procesos de desarrollo más actividades, más artefactos y más restricciones, basándose en los puntos débiles detectados. Sin embargo, el resultado final sería un proceso de desarrollo más complejo que puede incluso limitar la propia habilidad del equipo para llevar a cabo el proyecto. Otra aproximación es centrarse en otras dimensiones, como por ejemplo el factor humano o el producto software. Esta es la filosofía de las metodologías ágiles, las cuales dan mayor valor al individuo, a la colaboración con el cliente y al desarrollo incremental del software con iteraciones muy cortas. Este enfoque está mostrando su efectividad en proyectos con requisitos muy cambiantes y cuando se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad. Las metodologías ágiles están revolucionando la manera de producir software, y a la vez generando un amplio debate entre sus seguidores y quienes por escepticismo o convencimiento no las ven como alternativa para las metodologías tradicionales. En este trabajo se presenta resumidamente el contexto en el que surgen las metodologías ágiles, sus valores, principios y comparación con las metodologías tradicionales. Además se describen brevemente las principales propuestas, especialmente Programación Extrema (eXtreme Programming, XP) la metodología ágil más popular en la actualidad.

PALABRAS CLAVE. Procesos de Software, Metodologías Ágiles, Programación Extrema (XP)

1. INTRODUCCIÓN

En las dos últimas décadas las notaciones de modelado y posteriormente las herramientas pretendieron ser las "balas de plata" para el éxito en el desarrollo de software, sin embargo, las expectativas no fueron satisfechas. Esto se debe en gran parte a que otro importante elemento, la metodología de desarrollo, había sido postergado. De nada sirven buenas notaciones y herramientas si no se proveen directivas para su aplicación. Así, esta década ha comenzado con un creciente interés en metodologías de desarrollo. Hasta hace poco el proceso de desarrollo llevaba asociada un marcado énfasis en el control del proceso mediante una rigurosa definición de roles, actividades y artefactos, incluyendo modelado y documentación detallada. Este esquema "tradicional" para abordar el desarrollo de software ha demostrado ser efectivo y necesario en proyectos de gran tamaño (respecto a tiempo y recursos), donde por lo general se exige un alto grado de ceremonia en el proceso. Sin embargo, este enfoque no resulta ser el más adecuado para muchos de los proyectos actuales donde el entorno del sistema es muy cambiante, y en donde se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad. Ante las dificultades para utilizar metodologías tradicionales con estas restricciones de tiempo y flexibilidad, muchos equipos de desarrollo se resignan a prescindir del "buen hacer" de la ingeniería del software, asumiendo el riesgo que ello conlleva. En este escenario, las metodologías ágiles emergen como una posible respuesta para llenar ese vacío metodológico. Por estar especialmente orientadas para proyectos pequeños, las metodologías ágiles constituyen una solución a medida para ese entorno, aportando una elevada simplificación que a pesar de ello no renuncia a las prácticas esenciales para asegurar la calidad del producto.

Las metodologías ágiles son sin duda uno de los temas recientes en ingeniería de software que están acaparando gran interés. Prueba de ello es que se están haciendo un espacio destacado en

la mayoría de conferencias y workshops celebrados en los últimos años. Es tal su impacto que actualmente existen 4 conferencias internacionales de alto nivel y específicas sobre el tema[1]. Además ya es un área con cabida en prestigiosas revistas internacionales. En la comunidad de la ingeniería del software, se está viviendo con intensidad un debate abierto entre los partidarios de las metodologías tradicionales (referidas peyorativamente como "metodologías pesadas") y aquellos que apoyan las ideas emanadas del "Manifiesto Ágil"[2]. La curiosidad que siente la mayor parte de ingenieros de software, profesores, e incluso alumnos, sobre las metodologías ágiles hace prever una fuerte proyección industrial. Por un lado, para muchos equipos de desarrollo el uso de metodologías tradicionales les resulta muy lejano a su forma de trabajo actual considerando las dificultades de su introducción e inversión asociada en formación y herramientas. Por otro, las características de los proyectos para los cuales las metodologías ágiles han sido especialmente pensadas se ajustan a un amplio rango de proyectos industriales de desarrollo de software; aquellos en los cuales los equipos de desarrollo son pequeños, con plazos reducidos, requisitos volátiles, y/o basados en nuevas tecnologías.

El artículo está organizado como sigue. En la sección 2 se introducen las principales características de las metodologías ágiles, recogidas en el Manifiesto y se hace una comparación con las tradicionales. La sección 3 se centra en *eXtreme Programming* (XP), presentando sus características particulares, el proceso que se sigue y las prácticas que propone. En la sección 4 se citan otros métodos ágiles, enumerándose sus principales características. Finalmente aparecen las conclusiones.

2. METODOLOGÍAS ÁGILES

En febrero de 2001, tras una reunión celebrada en Utah-EEUU, nace el término "ágil" aplicado al desarrollo de software. En esta reunión participan un grupo de 17 expertos de la industria del software, incluyendo algunos de los creadores o impulsores de metodologías de software. Su objetivo fue esbozar los valores y principios que deberían permitir a los equipos desarrollar software rápidamente y respondiendo a los cambios que puedan surgir a lo largo del proyecto. Se pretendía ofrecer una alternativa a los procesos de desarrollo de software tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas.

Tras esta reunión se creó *The Agile Alliance*[3], una organización, sin ánimo de lucro, dedicada a promover los conceptos relacionados con el desarrollo ágil de software y ayudar a las organizaciones para que adopten dichos conceptos. El punto de partida es fue el Manifiesto Ágil, un documento que resume la filosofía "ágil".

2.1. El Manifiesto Ágil.

Según el Manifiesto se valora:

- **Al individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas.** La gente es el principal factor de éxito de un proyecto software. Es más importante construir un buen equipo que construir el entorno. Muchas veces se comete el error de construir primero el entorno y esperar que el equipo se adapte automáticamente. Es mejor crear el equipo y que éste configure su propio entorno de desarrollo en base a sus necesidades.
- **Desarrollar software que funciona más que conseguir una buena documentación.** La regla a seguir es "no producir documentos a menos que sean necesarios de forma inmediata para tomar un decisión importante". Estos documentos deben ser cortos y centrarse en lo fundamental.

[1] XP Agile Universe: www.agileuniverse.com Conference on eXtreme Programming and Agile Processes in Software Engineering: www.xp2004.org Agile Development Conference (EEUU): www.agiledevelopmentconference.com. Agile Development Conference (Australia): www.softed.com/adc2003/

[2] agilemanifesto.org

[3] www.agilealliance.com

- **La colaboración con el cliente más que la negociación de un contrato**. Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito.
- **Responder a los cambios más que seguir estrictamente un plan**. La habilidad de responder a los cambios que puedan surgir a los largo del proyecto (cambios en los requisitos, en la tecnología, en el equipo, etc.) determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta.

Los valores anteriores inspiran los doce principios del manifiesto. Son características que diferencian un proceso ágil de uno tradicional Los dos primeros principios son generales y resumen gran parte del espíritu ágil. El resto tienen que ver con el proceso a seguir y con el equipo de desarrollo, en cuanto metas a seguir y organización del mismo. Los principios son:

I. La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporte un valor.

II. Dar la bienvenida a los cambios. Se capturan los cambios para que el cliente tenga una ventaja competitiva.

III. Entregar frecuentemente software que funcione desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.

IV. La gente del negocio y los desarrolladores deben trabajar juntos a lo largo del proyecto.

V. Construir el proyecto en torno a individuos motivados. Darles el entorno y el apoyo que necesitan y confiar en ellos para conseguir finalizar el trabajo.

VI. El diálogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.

VII. El software que funciona es la medida principal de progreso.

VIII. Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.

IX. La atención continua a la calidad técnica y al buen diseño mejora la agilidad.

X. La simplicidad es esencial.

XI. Las mejores arquitecturas, requisitos y diseños surgen de los equipos organizados por sí mismos.

XII. En intervalos regulares, el equipo reflexiona respecto a cómo llegar a ser más efectivo, y según esto ajusta su comportamiento.

2.2. Comparación

La Tabla 1 recoge esquemáticamente las principales diferencias de las metodologías ágiles con respecto a las tradicionales ("no ágiles"). Estas diferencias que afectan no sólo al proceso en sí, sino también al contexto del equipo así como a su organización.

3. PROGRAMACIÓN EXTREMA (*EXTREME PROGRAMMING*, XP)

XP[4] [2] es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. XP se basa en realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. XP se define como especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.

[4] www extremeprogramming org, www.xprogramming com, c2. com/cgi/wiki?ExtremeProgramming

Metodologías Ágiles	Metodologías Tradicionales
Basadas en heurísticas provenientes de prácticas de producción de código	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo
Especialmente preparados para cambios durante el proyecto	Cierta resistencia a los cambios
Impuestas internamente (por el equipo)	Impuestas externamente
Proceso menos controlado, con pocos principios	Proceso mucho más controlado, con numerosas políticas/normas
No existe contrato tradicional o al menos es bastante flexible	Existe un contrato prefijado
El cliente es parte del equipo de desarrollo	El cliente interactúa con el equipo de desarrollo mediante reuniones
Grupos pequeños (<10 integrantes) y trabajando en el mismo sitio	Grupos grandes y posiblemente distribuidos
Pocos artefactos	Más artefactos
Pocos roles	Más roles
Menos énfasis en la arquitectura del software	La arquitectura del software es esencial y se expresa mediante modelos

Tabla 1. Diferencias entre metodologías ágiles y no ágiles

Los principios y prácticas son de sentido común pero llevadas al extremo, de ahí proviene su nombre. Kent Beck, el padre de XP, describe la filosofía de XP en [2] sin cubrir los detalles técnicos y de implantación de las prácticas. Posteriormente, otras publicaciones de experiencias se han encargado de dicha tarea. A continuación presentaremos las características esenciales de XP organizadas en los tres apartados siguientes: historias de usuario, roles, proceso y prácticas.

3.1. Las Historias de Usuario

Son la técnica utilizada para especificar los requisitos del software. Se trata de tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean requisitos funcionales o no funcionales. El tratamiento de las historias de usuario es muy dinámico y flexible. Cada historia de usuario es lo suficientemente comprensible y delimitada para que los programadores puedan implementarla en unas semanas [12].

Beck en su libro [2] presenta un ejemplo de ficha (*customer story and task card*) en la cual pueden reconocerse los siguientes contenidos: fecha, tipo de actividad (nueva, corrección, mejora), prueba funcional, número de historia, prioridad técnica y del cliente, referencia a otra historia previa, riesgo, estimación técnica, descripción, notas y una lista de seguimiento con la fecha, estado cosas por terminar y comentarios. A efectos de planificación, las historias pueden ser de una a tres semanas de tiempo de programación (para no superar el tamaño de una iteración). Las historias de usuario son descompuestas en tareas de programación (task card) y asignadas a los programadores para ser implementadas durante una iteración.

3.2. Roles XP

Los roles de acuerdo con la propuesta original de Beck son:

- **Programador.** El programador escribe las pruebas unitarias y produce el código del sistema.

- **Cliente.** Escribe las historias de usuario y las pruebas funcionales para validar su implementación. Además, asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración centrándose en aportar mayor valor al negocio.

- **Encargado de pruebas (*Tester*).** Ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para pruebas.

- **Encargado de seguimiento** (*Tracker*). Proporciona realimentación al equipo. Verifica el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, para mejorar futuras estimaciones. Realiza el seguimiento del progreso de cada iteración.

- **Entrenador** (*Coach*). Es responsable del proceso global. Debe proveer guías al equipo de forma que se apliquen las prácticas XP y se siga el proceso correctamente.

- **Consultor**. Es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto, en el que puedan surgir problemas.

- **Gestor** (*Big boss*). Es el vínculo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es de coordinación.

3.3. Proceso XP

El ciclo de desarrollo consiste (a grandes rasgos) en los siguientes pasos [12]:

1. El cliente define el valor de negocio a implementar.
2. El programador estima el esfuerzo necesario para su implementación.
3. El cliente selecciona qué construir, de acuerdo con sus prioridades y las restricciones de tiempo.
4. El programador construye ese valor de negocio.
5. Vuelve al paso 1.

En todas las iteraciones de este ciclo tanto el cliente como el programador aprenden. No se debe presionar al programador a realizar más trabajo que el estimado, ya que se perderá calidad en el software o no se cumplirán los plazos. De la misma forma el cliente tiene la obligación de manejar el ámbito de entrega del producto, para asegurarse que el sistema tenga el mayor valor de negocio posible con cada iteración.

El ciclo de vida ideal de XP consiste de seis fases [2]: Exploración, Planificación de la Entrega (*Release*), Iteraciones, Producción, Mantenimiento y Muerte del Proyecto.

3.4. Prácticas XP

La principal suposición que se realiza en XP es la posibilidad de disminuir la mítica curva exponencial del costo del cambio a lo largo del proyecto, lo suficiente para que el diseño evolutivo funcione. Esto se consigue gracias a las tecnologías disponibles para ayudar en el desarrollo de software y a la aplicación disciplinada de las siguientes prácticas.

- **El juego de la planificación**. Hay una comunicación frecuente el cliente y los programadores. El equipo técnico realiza una estimación del esfuerzo requerido para la implementación de las historias de usuario y los clientes deciden sobre el ámbito y tiempo de las entregas y de cada iteración.
- **Entregas pequeñas**. Producir rápidamente versiones del sistema que sean operativas, aunque no cuenten con toda la funcionalidad del sistema. Esta versión ya constituye un resultado de valor para el negocio. Una entrega no debería tardar más 3 meses.
- **Metáfora**. El sistema es definido mediante una metáfora o un conjunto de metáforas compartidas por el cliente y el equipo de desarrollo. Una metáfora es una historia compartida que describe cómo debería funcionar el sistema (conjunto de nombres que actúen como vocabulario para hablar sobre el dominio del problema, ayudando a la nomenclatura de clases y métodos del sistema).
- **Diseño simple**. Se debe diseñar la solución más simple que pueda funcionar y ser implementada en un momento determinado del proyecto.
- **Pruebas**. La producción de código está dirigida por las pruebas unitarias. Éstas son son establecidas por el cliente antes de escribirse el código y son ejecutadas constantemente ante cada modificación del sistema.

- **Refactorización (*Refactoring*).** Es una actividad constante de reestructuración del código con el objetivo de remover duplicación de código, mejorar su legibilidad, simplificarlo y hacerlo más flexible para facilitar los posteriores cambios. Se mejora la estructura interna del código sin alterar su comportamiento externo [8].
- **Programación en parejas.** Toda la producción de código debe realizarse con trabajo en parejas de programadores. Esto conlleva ventajas implícitas (menor tasa de errores, mejor diseño, mayor satisfacción de los programadores, ...).
- **Propiedad colectiva del código.** Cualquier programador puede cambiar cualquier parte del código en cualquier momento.
- **Integración continua.** Cada pieza de código es integrada en el sistema una vez que esté lista. Así, el sistema puede llegar a ser integrado y construido varias veces en un mismo día.
- **40 horas por semana.** Se debe trabajar un máximo de 40 horas por semana. No se trabajan horas extras en dos semanas seguidas. Si esto ocurre, probablemente está ocurriendo un problema que debe corregirse. El trabajo extra desmotiva al equipo.
- **Cliente in-situ.** El cliente tiene que estar presente y disponible todo el tiempo para el equipo. Éste es uno de los principales factores de éxito del proyecto XP. El cliente conduce constantemente el trabajo hacia lo que aportará mayor valor de negocio y los programadores pueden resolver de manera inmediata cualquier duda asociada. La comunicación oral es más efectiva que la escrita.
- **Estándares de programación.** XP enfatiza que la comunicación de los programadores es a través del código, con lo cual es indispensable que se sigan ciertos estándares de programación para mantener el código legible.

El mayor beneficio de las prácticas se consigue con su aplicación conjunta y equilibrada puesto que se apoyan unas en otras. Esto se ilustra en la Figura 1 (obtenida de [2]), donde una línea entre dos prácticas significa que las dos prácticas se refuerzan entre sí. La mayoría de las prácticas propuestas por XP no son novedosas sino que en alguna forma ya habían sido propuestas en ingeniería del software e incluso demostrado su valor en la práctica (ver [1] para un análisis histórico de ideas y prácticas que sirven como antecedentes a las utilizadas por las metodologías ágiles). El mérito de XP es integrarlas de una forma efectiva y complementarlas con otras ideas desde la perspectiva del negocio, los valores humanos y el trabajo en equipo.

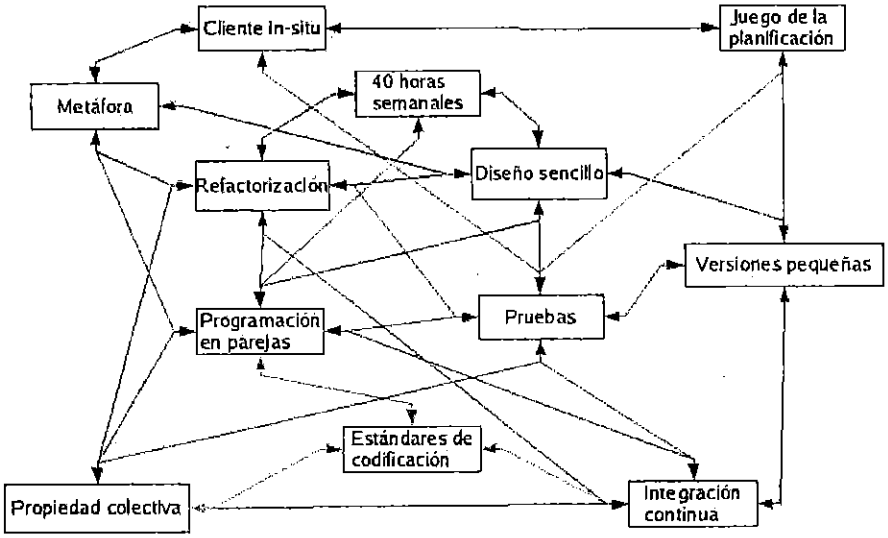


Figura 1. Las prácticas se refuerzan entre sí

3. OTRAS METODOLOGÍAS ÁGILES

Aunque los creadores e impulsores de las metodologías ágiles más populares han suscrito el manifiesto ágil y coinciden con los principios enunciados anteriormente, cada metodología tiene características propias y hace hincapié en algunos aspectos más específicos. A continuación se resumen otras metodologías ágiles. La mayoría de ellas ya estaban siendo utilizadas con éxito en proyectos reales pero les faltaba una mayor difusión y reconocimiento.

- **SCRUM**[5] [16]. Desarrollada por Ken Schwaber, Jeff Sutherland y Mike Beedle. Define un marco para la gestión de proyectos, que se ha utilizado con éxito durante los últimos 10 años. Está especialmente indicada para proyectos con un rápido cambio de requisitos. Sus principales características se pueden resumir en dos. El desarrollo de software se realiza mediante iteraciones, denominadas *sprints*, con una duración de 30 días. El resultado de cada *sprint* es un incremento ejecutable que se muestra al cliente. La segunda característica importante son las reuniones a lo largo proyecto, entre ellas destaca la reunión diaria de 15 minutos del equipo de desarrollo para coordinación e integración.

- **Crystal Methodologies**[6] [5]. Se trata de un conjunto de metodologías para el desarrollo de software caracterizadas por estar centradas en las personas que componen el equipo y la reducción al máximo del número de artefactos producidos. Han sido desarrolladas por Alistair Cockburn. El desarrollo de software se considera un juego cooperativo de invención y comunicación, limitado por los recursos a utilizar. El equipo de desarrollo es un factor clave, por lo que se deben invertir esfuerzos en mejorar sus habilidades y destrezas, así como tener políticas de trabajo en equipo definidas. Estas políticas dependerán del tamaño del equipo, estableciéndose una clasificación por colores, por ejemplo Crystal Clear (3 a 8 miembros) y Crystal Orange (25 a 50 miembros).

- **Dynamic Systems Development Method**[7] **(DSDM)** [17]. Define el marco para desarrollar un proceso de producción de software. Nace en 1994 con el objetivo de crear una metodología RAD unificada. Sus principales características son: es un proceso iterativo e incremental y el equipo de desarrollo y el usuario trabajan juntos. Propone cinco fases: estudio viabilidad, estudio del negocio, modelado funcional, diseño y construcción, y finalmente implementación. Las tres últimas son iterativas, además de existir realimentación a todas las fases.

- **Adaptive Software Development**[8] **(ASD)** [9]. Su impulsor es Jim Highsmith. Sus principales características son: iterativo, orientado a los componentes software más que a las tareas y tolerante a los cambios. El ciclo de vida que propone tiene tres fases esenciales: especulación, colaboración y aprendizaje. En la primera de ellas se inicia el proyecto y se planifican las características del software; en la segunda desarrollan las características y finalmente en la tercera se revisa su calidad, y se entrega al cliente. La revisión de los componentes sirve para aprender de los errores y volver a iniciar el ciclo de desarrollo.

- **Feature-Driven Development**[9] **(FDD)** [3]. Define un proceso iterativo que consta de 5 pasos. Las iteraciones son cortas (hasta 2 semanas). Se centra en las fases de diseño e implementación del sistema partiendo de una lista de características que debe reunir el software. Sus impulsores son Jeff De Luca y Peter Coad.

- **Lean Development**[10] **(LD)** [15]. Definida por Bob Charette's a partir de su experiencia en proyectos con la industria japonesa del automóvil en los años 80 y utilizada en numerosos proyectos de telecomunicaciones en Europa. En LD, los cambios se consideran riesgos, pero si se manejan adecuadamente se pueden convertir en oportunidades que mejoren la

[5] www.controlchaos.com
[6] www.crystalmethodologies.org
[7] www.dsdm.org
[8] www.adaptivesd.com
[9] www.featuredrivendevelopment.com
[10] www.poppendieck.com

productividad del cliente. Su principal característica es introducir un mecanismo para implementar dichos cambios.

4. CONCLUSIONES

No existe una metodología universal para hacer frente con éxito a cualquier proyecto de desarrollo de software. Toda metodología debe ser adaptada al contexto del proyecto (recursos técnicos y humanos, tiempo de desarrollo, tipo de sistema, etc. Históricamente, las metodologías tradicionales han intentado abordar la mayor cantidad de situaciones de contexto del proyecto, exigiendo un esfuerzo considerable para ser adaptadas, sobre todo en proyectos pequeños y con requisitos muy cambiantes. Las metodologías ágiles ofrecen una solución casi a medida para una gran cantidad de proyectos que tienen estas características. Una de las cualidades más destacables en una metodología ágil es su sencillez, tanto en su aprendizaje como en su aplicación, reduciéndose así los costos de implantación en un equipo de desarrollo. Esto ha llevado hacia un interés creciente en las metodologías ágiles. Sin embargo, hay que tener presente una serie de inconvenientes y restricciones para su aplicación, tales como: están dirigidas a equipos pequeños o medianos (Beck sugiere que el tamaño de los equipos se limite de 3 a 20 como máximo, otros dicen no más de 10 participantes), el entorno físico debe ser un ambiente que permita la comunicación y colaboración entre todos los miembros del equipo durante todo el tiempo, cualquier resistencia del cliente o del equipo de desarrollo hacia las prácticas y principios puede llevar al proceso al fracaso (el clima de trabajo, la colaboración y la relación contractual son claves), el uso de tecnologías que no tengan un ciclo rápido de realimentación o que no soporten fácilmente el cambio, etc.

Falta aún un cuerpo de conocimiento consensuado respecto de los aspectos teóricos y prácticos de la utilización de metodologías ágiles, así como una mayor consolidación de los resultados de aplicación. La actividad de investigación está orientada hacia líneas tales como: métricas y evaluación del proceso, herramientas específicas para apoyar prácticas ágiles, aspectos humanos y de trabajo en equipo. Entre estos esfuerzos destacan proyectos como NAME[11] (*Network for Agile Methodologies Experience*) en el cual hemos participado como nodo en España.

Aunque en la actualidad ya existen libros asociados a cada una de las metodologías ágiles existentes y también abundante información en internet, es XP la metodología que resalta por contar con la mayor cantidad de información disponible y es con diferencia la más popular.

BIBLIOGRAFIA

[1] Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J. "Agile software development methods Review and analysis". VTT Publications. 2002.

[2] Beck, K.. "Extreme Programming Explained. Embrace Change", Pearson Education, 1999. Traducido al español como: "Una explicación de la programación extrema. Aceptar el cambio", Addison Wesley, 2000.

[3] Coad P., Lefebvre E., De Luca J. "Java Modeling In Color With UML: Enterprise Components and Process". Prentice Hall. 1999.

[4] Cockbun, A. "Agile Software Development". Addison-Wesley. 2001.

[5] Fowler, M., Beck, K., Brant, J. "Refactoring: Improving the Design of Existing Code". Addison-Wesley. 1999

[6] Highsmith J., Orr K. "Adaptive Software Development: A Collaborative Approach to Managing Complex Systems". Dorset House. 2000.

[7] Jeffries, R., Anderson, A., Hendrickson, C. "Extreme Programming Installed". Addison-Wesley. 2001

[8] Poppendieck M., Poppendieck T. "Lean Software Development: An Agile Toolkit for Software Development Managers". Addison Wesley. 2003.

[9] Schwaber K., Beedle M., Martin R.C. "Agile Software Development with SCRUM". Prentice Hall. 2001.

[10] Stapleton J. "Dsdm Dynamic Systems Development Method: The Method in Practice". Addison-Wesley. 1997.

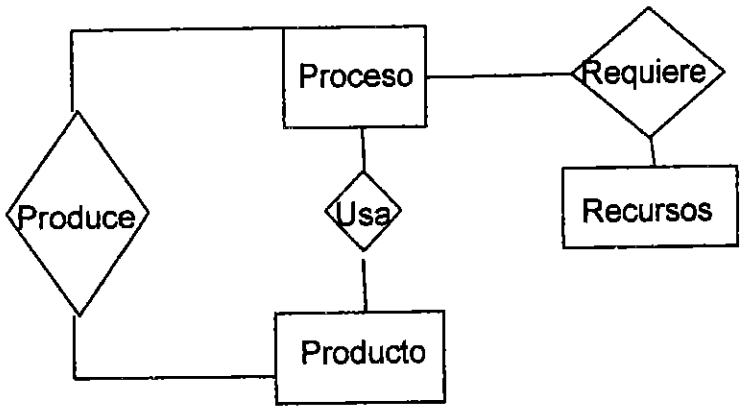
[11] name case.unibz.it/

MEDICIONES DE SOFTWARE

Entidades y atributos

- **Procesos**: Colecciones de actividades de software relacionadas
- **Productos:** Artefactos, entregable documentos que resultan de una actividad de proceso.
- **Recursos:** Entidades requeridas para una actividad del proceso.

Modelo de Medición



Tipificación de los atributos del software

Los atributos del software se clasifican en:

Internos: Son del producto, proceso o recurso (PPoR) que pueden medirse en términos del PPoR mismo. Se mide examinando la entidad independientemente de su comportamiento.

Externos: Son los atributos del PPoR que se miden en términos de cómo el PPoR se relaciona con su entorno. El comportamiento del PPoR es más importante que él mismo.

Mediciones de Productos

Entidades (Productos)	Atributos Internos	Atributos Externos
Especificaciones	Tamaño, reuso, modularidad, redundancia, funcionalidad, correctitud sintáctica	Compresibilidad mantenibilidad
Diseño	Tamaño, reuso, modularidad, acoplamiento, cohesión, funcionalidad	Calidad, complejidad, mantenibilidad
Código	Tamaño, reuso, modularidad, acoplamiento, funcionalidad, complejidad algorítmica, estructura de flujo de control	Confiabilidad, usabilidad, mantenibilidad
Datos de prueba	Tamaño, nivel del cobertura	Calidad

Mediciones de Procesos

Entidades (Procesos)	Atributos Internos	Atributos Externos
Especificación de Construcción	Tiempo, esfuerzo, número de cambios de requerimientos	Calidad, costo, estabilidad
Diseño detallado	Tiempo, esfuerzo, número de errores de especificación encontrados	Costo, costo-efectividad
Testeo o Prueba	Tiempo, esfuerzo, número de errores de codificación encontrados	Costo, costo-efectividad, estabilidad

Mediciones de Recursos

Entidades (Recursos)	Atributos Internos	Atributos Externos
Personal	Edad, precio	Productividad, experiencia, inteligencia
Equipos	Tamaño, nivel de comunicación, estructuración	Productividad, calidad
Software	Precio, tamaño	Usabilidad, confiabilidad
Hardware	Precio, velocidad, tamaño de memoria	Confiabilidad
Oficinas	Tamaño, temperatura, luz	Calidad, confort

Nuestro objetivo es estudiar y medir los atributos internos de los productos. Una única medida sería útil para alguno pero no para otro objetivo. Tenemos como meta medir el tamaño del software. Según el esquema de Fenton-Pfleeger:

> **Cada atributo debe capturar un aspecto clave del tamaño del software.**

El tamaño del software se puede describir con:

- **Longitud**: Tamaño físico del producto
- **Funcionalidad**: Funciones que provee al usuario
- **Complejidad Computacional o del problema**: Que a su vez se puede clasificar en:
  - ***Algorítmica***: Mide la eficiencia del software
  - ***Estructural***: Mide la estructura del software que implementa el algoritmo
  - ***Cognitiva***: Mide el esfuerzo requerido para entender el software.
- **Reuso**: Uso de partes pre-existentes. Mide cuánto se reusó de una versión previa de un producto pre-existente. Es una medida clase cuando el tamaño es entrada en un modelo de esfuerzo, costo y productividad.

MEDIDAS DE LONGITUD DEL SOFTWARE

- **Código**: El código se mide en **LOC** (Líneas de código), aunque hay problemas en cuanto a cómo considerar las líneas en blanco y los comentarios. Si para medir tamaño se consideran solamente las líneas de código en realidad estaríamos midiendo solamente líneas no comentadas (tampoco mediríamos espacio en disco ni páginas de listado). Las líneas de código pueden asociarse al esfuerzo pero de alguna manera estaríamos diciendo que los comentarios no requieren esfuerzo. Debido a estas razones, Fenton/Pfleeger propone medir:

Longitud Total(LOC) = NCLOC + CLOC

Donde:

NCLOC: Líneas de código no comentadas

CLOC: Líneas de código comentadas.

El SEI (Software Engineering Institute) propone el siguiente enfoque para hacer las cuentas. Elegir entre:

- Clasificar las líneas de código: Ejecutable, declaración, directivas del compilador, comentarios y líneas en blanco.

- Analizar cómo se produce el código: generado, convertido, modificado o removido.

- Orígen del código: Nuevo, adaptado de otro trabajo.

Existen otras definiciones de longitud de código, pero son menos usadas:

- Longitud: Cantidad de bytes requeridos para almacenar el programa.

- Longitud: Cantidad de caracteres en el texto del programa.

- **Especificaciones y diseño**: Su tamaño se mide en cantidad de páginas, aunque una especificación o un diseño consta de texto y gráficos que son inconmensurables.

Predicción de longitud

De acuerdo a estudios realizados sobre proyectos completos, resultados empíricos, se puede predecir que:

$$D = 49 * L^{1.01}$$

Donde:

D: Longitud de la documentación en páginas

L: Longitud del programa medida en KLOC

REUSO

Abarca requerimientos, diseños, documentación, datos de testeo y código. Sus grandes ventajas es que hay una mayor productividad, una mayor calidad en los productos y que permite a los desarrolladores concentrarse solamente en lo nuevo que hay que producir.

Funcionalidad

Además de por su tamaño, el software se puede medir por su funcionalidad. Este concepto captura la noción intuitiva de "cantidad de función" del producto o la descripción de cómo debe ser el producto. Algunos ejemplos de modelos que miden funcionalidad son:

- Puntos de Función de Albrecht

- Peso de la especificación de De Marco

15.2El proceso de gestión del proyecto

Este proceso constituye el **primer nivel del proceso de Ingeniería de Software**. ¿Cuáles son los elementos clave de la gestión del Proyecto de Software?

Comienzo del proyecto: Debe establecerse el ámbito y los objetivos, deben considerarse soluciones alternativas y deben identificarse restricciones técnicas y de gestión ya que sin ésta información es imposible hacer estimaciones de costos razonables, una identificación realista de las tareas del proyecto o un plan de trabajo adecuado que proporcione una indicación significativa del progreso del proyecto.

La definición del ámbito y los objetivos se hacen de común acuerdo entre el desarrollador y el cliente.

objetivos: indican las funciones globales del proyecto
ámbito: identifica las funciones primordiales que debe llevar a cabo el software e intenta limitar esas funciones de manera cuantitativa.

Medición y Métricas: El proceso se mide para intentar mejorarlo. El producto se mide para aumentar la calidad.

Proporcionan a los gestores y técnicos una mejor comprensión del proceso de Ingeniería de Software y del producto que se genera.

Estimación: Una de las actividades cruciales del proceso de gestión de proyectos de software es la **planificación.** Cuando se planifica un proyecto de software se tienen que obtener estimaciones del esfuerzo humano requerido, de la duración cronológica del proyecto y del costo.

Muchas veces, las estimaciones se **hacen basándose en experiencias pasadas como única guía.** Es decir, si un proyecto es bastante parecido a uno anterior en tamaño y función, es probable que requiera aproximadamente la misma cantidad de esfuerzo, que dure aproximadamente el mismo tiempo y que cueste aproximadamente los mismo que el anterior. Pero, qué sucede si el proyecto es totalmente distinto?

Se han desarrollado varias técnicas de estimación para el desarrollo de software, cada una con ventajas y desventajas. Todas, sin embargo, tienen en común los siguientes atributos:

a) El ámbito del proyecto se debe establecer de antemano.
b) Se usan las métricas del software como base para hacer estimaciones.
c) El proyecto se desglosa en partes más pequeñas que se estiman individualmente.

Análisis de riesgos: Cada vez que se va a construir un programa surgen ciertas **áreas de incertidumbre:** ¿Se entienden realmente las necesidades del cliente? ¿Se podrán implementar antes de la fecha tope las funciones que se tienen

que realizar? ¿Se encontrarán problemas técnicos de difícil solución que en este momento no son aparentes? ¿La planificación será destruida por los cambios que invariablemente aparecen en cualquier proyecto?.

El análisis de riesgos es vital para una buena gestión del proyecto, aunque se emprendan muchos de ellos sin haber considerado los riesgos concretos.

El análisis de riesgos **consiste en una serie de pasos de "control de riesgos" que nos permiten "combatirlos"**: identificación de riesgos, cálculo de riesgos, priorización de riesgos y supervisión de riesgos. Estos pasos se aplican a lo largo de todo el proceso de ingeniería de Software.

Planificación temporal: ¿Se ha establecido la agenda sobre la marcha, o se ha planificado por adelantado? ¿Se ha realizado el trabajo según ha sido necesario o se ha identificado previamente un conjunto de tareas bien definidas? ¿Se han identificado los gestores únicamente en la fecha tope, o se ha identificado un camino crítico y se ha supervisado para asegurar que se puedan cumplir los plazos? ¿El progreso se ha medido por lo que está hecho o se ha establecido un conjunto de hitos igualmente espaciado?

En realidad la planificación **consiste en identificar una serie de tareas del proyecto**, establecer interdependencia entre esas tareas, estimar el esfuerzo asociado con cada tarea, hacer una asignación de personal y otros recursos, crear una red de tareas y desarrollar una agenda de fechas.

Seguimiento y Control: Una vez que se ha definido la agenda de desarrollo, comienza la actividad de seguimiento y control. El gestor del proyecto sigue la pista de cada tarea establecida en la agenda. Si una tarea se sale de la agenda puede reasignar recursos, reordenar tareas, modificar los compromisos de entrega para resolver el problema.

15.3Métricas para la productividad y la calidad del software

Las métricas del software se refieren **a** un amplio espectro de medidas para el software de computadoras. Las métricas de productividad **y** calidad nos interesan durante la planificación del proyecto **y son medidas del rendimiento de la "salida" del desarrollo de software como función del esfuerzo aplicado.**

15.3.1 Medición del software

Existen varias razones por las que medir el software:

i. Para indicar la calidad del producto.
ii. Para evaluar la productividad de la gente que lo desarrolla
iii. Para evaluar los beneficios derivados del uso de métodos y herramientas de Ingeniería de Software (en términos de productividad y calidad).
iv. Para establecer una línea de base para la estimación
v. Para ayudar a justificar el uso de nuevas herramientas o de formación adicional.

Las mediciones del mundo físico se dividen en **medidas directas** (ej: la longitud de un tornillo) ***y* medidas indirectas** (ej: la calidad de los tornillos producidos medida por el porcentaje de tornillos rechazados).

Desde el punto de vista del software, se clasifican en:
Medidas directas del proceso de I.S..: el costo y el esfuerzo aplicado

Medidas directas del producto: Líneas de código producidas, velocidad de ejecución, el tamaño de memoria y los defectos observados en un determinado período de tiempo.

Medidas indirectas del producto*:* funcionalidad, calidad, complejidad, eficiencia, facilidad de mantenimiento, etc.

Aunque el costo, el esfuerzo requerido para construir el software, el número de líneas de código y otras medidas directas son fáciles de obtener, **la calidad y la funcionalidad del software, o su eficiencia y facilidad de mantenimiento son más difíciles de evaluar** y sólo se pueden medir indirectamente.

Podemos hacer las siguientes **clasificaciones de las métricas del software:**

Métricas del software (Clasif.I)
- Métricas técnicas
- Métricas de calidad
- Métricas de productividad

Métricas del software (Clasif. II)
- Métricas orientadas al tamaño
- Métricas orientadas a la función
- Métricas orientadas a la persona

Las **métricas de productividad** se centran en el rendimiento del proceso de Ingeniería de Software.

Las **métricas de calidad** proporcionan una indicación de cómo se ajusta el software a los requisitos implícitos y explícitos del cliente.

Las **métricas técnicas** se centran en las características del software (por ejemplo: las complejidad lógica o el grado de modularidad).

En la **otra clasificación:**

Las **métricas orientadas al tamaño** se usan para obtener medidas directas

del resultado y de la calidad de la Ingeniería de Software.

Las **métricas orientadas a la función** proporcionan medidas indirectas.

Las **métricas orientadas a la persona** proporcionan información sobre la forma en que la gente desarrolla software y sobre el punto de vista humano de la efectividad de la herramientas y métodos.

15.3.1.1 Métricas orientadas al tamaño

Son **medidas directas** del software y del proceso por el cual se desarrolla. Si una organización de software mantiene registros sencillos se puede crear una tabla de datos orientados al tamaño que contenga, para cada proyecto de software realizado los siguientes datos:

Identificación del proyecto
Miles de líneas de código (KLDC)
Esfuerzo requerido (en personas-mes)
Costo (en pesos o dólares)
Cantidad de páginas de documentación
Cantidad de errores detectados en el primer año de uso
Cantidad de personas que trabajaron en el desarrollo del proyecto.

Debe tenerse en cuenta que las medidas de esfuerzo requerido y de costo registrados **se deben incluir todas las actividades de Ingeniería de Software** (análisis, diseño, codificación y prueba) y no sólo la codificación.

Con estos rudimentarios datos obtenidos de la tabla de datos se puede desarrollar, para cada proyecto, un **conjunto de métricas sencillas de productividad y de calidad orientadas al tamaño.** También se pueden calcular valores promedios de todos los proyectos. Por ejemplo:

Productividad = KLDC / personas-mes
Calidad = errores / KLOC
Coste = dólares / KLDC
Documentación = páginas de documentación / KLDC

Las **métricas orientadas al tamaño son bastante polémicas** y no están aceptadas universalmente como el mejor modo de medir el proceso de desarrollo de software. Esto se debe a usar líneas de código como medida clave, ya que, a pesar de que se calculan mediante un artificio sencillo para todos los proyectos de software, que muchos modelos de estimación de software existentes las usan como clave de entrada, son dependientes del lenguaje de programación que se use, perjudica a los programas más cortos pero bien diseñados, no se pueden adaptar fácilmente a los lenguajes no procedimentales y su utilización en la estimación requiere un nivel de detalle que puede ser difícil de conseguir (Por ejemplo: el planificador debe estimar las LDC a producir antes de completar el análisis y diseño).

15.3.1.2 Métricas orientadas a la función

Son **medidas indirectas del software y del proceso** por el cual se desarrolla. Se centran en la ***"funcionalidad"*** o ***"utilidad"*** del programa. Estas métricas fueron propuestas por Albrecht (1979) quien sugirió un acercamiento a la medida de productividad denominado **método del punto de función.** En resúmen, miden la cantidad de funcionalidad de un sistema descripto en una especificación (independientemente del modelo o técnica particular).

Los puntos de función (Pfs) se obtienen usando una relación empírica basada en medidas cuantitativas del domino de información del software y valoraciones subjetivas de la complejidad del software.

Estos puntos de función se calculan **rellenando la siguiente tabla.**

		Factor de peso		
Parámetro de medida	Cuenta	Simple	Medio	Complejo
Número de entradas de usuario (Inputs externos provistos por el usuario)	.....	X3	4	6=.....
Número de salidas de usuario (informes y mensajes)	.....	X4	5	7=.....
Número de peticiones al usuario (Inputs interactivos)	.....	X3	4	6=.....
Número de archivos (Maestros lógicos)	.....	X7	10	15=.....
Número de interfaces externos (Interfaces con otros sistemas)	.....	X5	7	10=.....
Cuenta-total	⟶			.....

Se determinan cinco características del ámbito de la información y los cálculos aparecen indicados en la posición apropiada de la tabla. Los valores del ámbito de la información están definidos de la siguiente manera:

Número de entradas de usuario: Se cuenta cada entrada de usuario que proporciona al software diferentes datos orientados a la aplicación. Las entradas deben distinguirse de las peticiones que se contabilizan por separado.

Número de salidas de usuario: Se cuenta cada salida que proporciona al usuario información orientada a la aplicación (informes, pantallas, mensajes de error, etc. No se cuentan los elementos de datos individuales dentro de un informe).

Número de peticiones al usuario: Son entradas interactivas que resultan de la

generación de algún tipo de respuesta en forma de salida interactiva.
Número de archivos: Se cuenta cada archivo maestro lógico (es decir una agrupación lógica de datos que puede ser parte de una gran base datos o un archivo independiente).
Número de interfaces externas: Se cuentan todas las interfaces legibles por la máquina (archivos de datos en cinta o disco) que son usados para transmitir información a otro sistema.

Cuando estos datos son recogidos se asocia un **valor de complejidad** a cada cuenta. Las organizaciones que usan métodos de puntos de función desarrollan criterios para determinar si una entrada determinada es simple, media o compleja. Pero la determinación de la complejidad es algo subjetivo.

Para calcular los puntos de función se usa la siguiente relación:

PF =	**UFC**	*	**TCF**
Cantidad	Cantidad de Puntos de Función no Ajustados		Factor de complejidad técnica

Donde:

$$\mathbf{UFC} = \Sigma \text{ (nro. Items de tipo}_i * \text{peso}_i) \quad \text{para i de 1 a 15}$$

Y

$$\mathbf{TCF} = 0{,}65 + 0{,}01 \times \Sigma(F_i) \quad \text{para i de 1 a 14}$$

donde:
Fi = 0 a 5 según el peso del factor (0: irrelevante, 3: medio, 5: esencial)

Y los factores son:

1 – Backup y recuperación confiable
2 – Data comunications
3 – Funciones distribuidas
4 – Performance
5 – Peso de la configuración
6 – Data entry en línea
7 – Fácil operación
8 – Actualización on-line
9 – Interface compleja
10 – Procesamiento complejo
11 – Reusabilidad
12 – Fácil instalación
13 – Múltiples sites
14 – Cambio facilitado

Los valores constantes de esta ecuación y los factores de peso aplicados a la cuentas de los ámbitos de información han sido determinados empíricamente.

Los PF se usan para definir cuándo y qué someter a reingeniería; para estimar casos de testeo; para calcular costo del software; para estimar costos, cronograma y esfuerzo del proyecto; para comprender los costos de mantenimiento;

para negociar contratos o para desarrollar métricas estándar.

Una vez calculados los puntos de función, se usan de forma análoga a las **LDC como medida de productividad,** calidad y otros atributos del software:

Productividad = PF / personas-mes
Calidad = errores / PF
Coste = dólares / PF
Documentación = páginas de documentación / PF

Esta **métrica es controvertida también,** ya que, a pesar de ser independiente del lenguaje de programación, y que está basada en datos que son más probables de ser conocidos al principio de la evolución del proyecto, se requiere una cierta "destreza" porque los cálculos están basados en la subjetividad de los datos; la información del ámbito de la información puede ser difícil de adquirir a posteriori y el PF no tiene un significado físico directo - es sólo un número.

Además de influir la subjetividad del factor tecnológico hay que llevar una doble contabilidad (UFC y TCF), ya dijimos que requiere una especificación completa del sistema y no son independientes del modelo de análisis y diseño. Existen problemas con el dominio de aplicación y con la teoría de la medida.

Peso de la especificación de De Marco

Brevemente, se trata de calcular:

- Function Bang: Nro. De primitivas funcionales (Burbujas de menor nivel)
- Data Bang: Nro. De entidades del DER.

Ambas medidas son fácilmente extendibles a otros enfoques del análisis estructurado.

18. Métricas técnicas para Sistemas Orientados a Objetos

Los principales objetivos de las métricas OO, al igual que para el software convencional, son:

- Comprender mejor la calidad del producto
- Estimar la efectividad del producto
- Mejorar la calidad del trabajo realizado en el nivel del proyecto.

Las medidas para cualquier producto de la ingeniería están gobernadas por las **características únicas de este producto**. Por esta razón, las métricas técnicas para sistemas OO deben ajustarse a las características que distinguen el software OO del software convencional.

Berard (95) define 5 características que dan lugar a unas métricas especializadas:

1. **Localización:** característica del software que indica la forma en que se concentra la información dentro de un programa.
   En el software convencional, ya que las funciones son los mecanismos de localización, las métricas se centran en la estructura interna o complejidad de las funciones.
   En los sistemas OO la localización está basada en objetos ya que la clase constituye la unidad básica. Por lo tanto las métricas deberían ser aplicables a la clase (objeto) como si se tratara de una entidad completa. Además, las métricas que reflejan la forma en que colaboran las clases deben poder adaptarse a las relaciones **uno – a – *muchos*** y **muchos – a – uno** ya que la relación entre operaciones (funciones) y clases no es necesariamente **uno – a – uno.**

2. **Encapsulamiento:** es el empaquetamiento de una colección de elementos.
   Para el software convencional se encuentran ejemplos de encapsulamiento de bajo nivel en registros, matrices y subprogramas.
   Para sistemas OO, el encapsulamiento abarca las responsabilidades de una clase, atributos y operaciones, y los estados de la clase, según se definen mediante valores específicos de los atributos.

3. **Ocultamiento de Información:** suprime (u oculta) los detalles operativos de un componente de programa.

Aquellas métricas que proporcionen un indicativo del grado en que se ha logrado el ocultamiento proporcionan un indicativo de la calidad del diseño OO.

4. **Herencia:** es un mecanismo que hace posible que las responsabilidades de un objeto se propague a otros objetos a lo largo de todos los niveles de la jerarquía de clases.
Las métricas para software OO se centran en la herencia contando el número de descendientes de una clase, el número de predecesores y el grado de anidamiento de la jerarquía de clases.

5. **Abstracción:** es un mecanismo que permite al diseñador centrarse en los detalles esenciales de algún componente de programa sin preocuparse de los detalles del nivel inferior.
Ya que una clase es una abstracción que se puede visualizar con muchos niveles distintos de detalle, las métricas OO, representan las abstracciones en términos de medidas de una clase.

Las métricas técnicas se pueden aplicar al modelo de análisis y al modelo de diseño. Las métricas que estudiaremos proporcionan información de calidad en el nivel de la clase OO y de las operaciones. También hay métricas aplicables a la gestión y comprobación de proyectos.

18.1 Métricas Orientadas a Clases

La clase es la unidad fundamental de todo sistema OO. Por ello, las medidas y métricas para una clase individual, la jerarquía de clases y las colaboraciones de clases son muy valiosas para un ingeniero de software que tenga que estimar la calidad de un diseño.

Todas las características de una clase (encapsula operaciones y atributos, herencia, colabora con otras clases) se pueden usar como base para una medida.

18.2 Métricas CK

Conjunto de métricas propuesto por Chidamer y Kemener (94) que son métricas de diseño basadas en clases para sistemas OO.

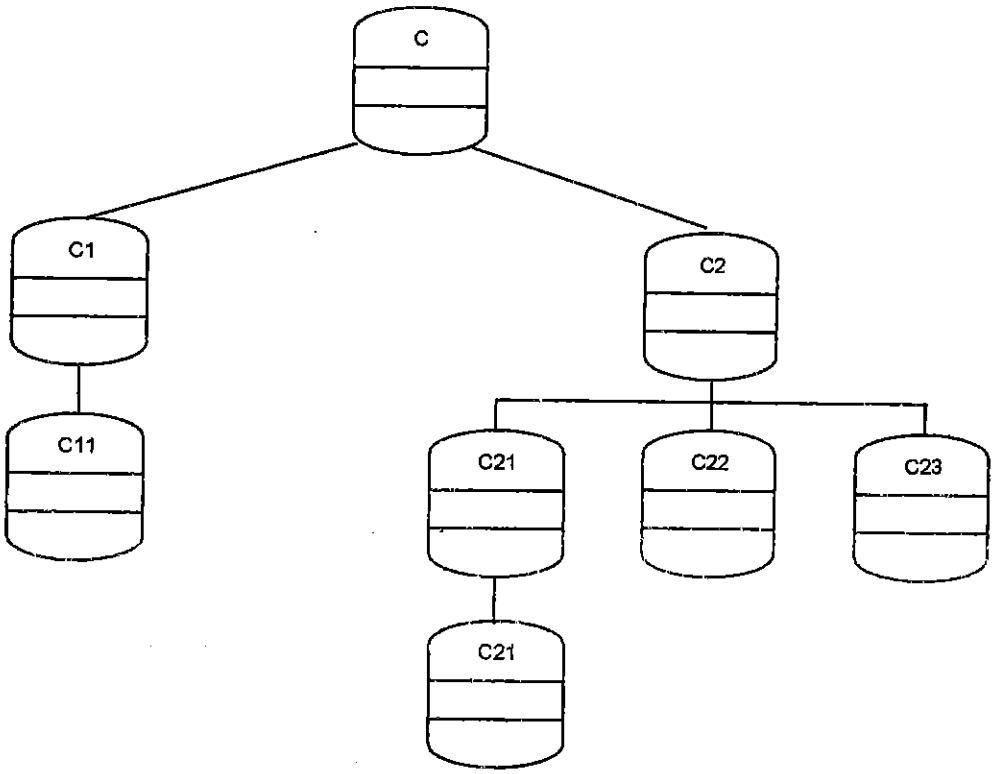
Métodos ponderados por clase (MPC): suponga que se definen, para la clase c, n métodos de complejidad c_1, c_2,..., c_n .

La métrica de complejidad específica que se relaciona (por ej.: la complejidad ciclomática (recuento de condiciones booleanas)) debe normalizarse de tal modo que la complejidad nominal para un método tome el valor 1, 0.

$$MPC = \Sigma c_i \qquad \text{para } i=1,...,n$$

El número de métodos y su complejidad es un indicador razonable de la cantidad de esfuerzo necesario para implementar y comprobar una clase.
A mayor cantidad de métodos, más complejo será el árbol de herencia. Así también, cada métodos es más específico de la aplicación limitando su potencial de reutilización.
Entonces, MPC debería mantener un valor tan bajo como sea razonable.

Árbol de profundidad de herencia (APH): esta métrica se define como "la longitud máxima desde el nodo hasta la raíz del árbol".
Tomemos como ejemplo la siguiente jerarquía de clases:



Para esta jerarquía de clases el valor de APH es 4.

A medida que APH crece, es más probable que las clases de niveles inferiores hereden muchos métodos. Esto puede producir dificultades cuando se intenta predecir el comportamiento de una clase.

Una jerarquía de clases profunda (con valor de APH grande) significa también una mayor complejidad de diseño. Pero también, los valores grandes de APH implican que se puedan reutilizar muchos métodos.

Número de descendientes (NDD): las subclases que son inmediatamente subordinadas a una clase en la jerarquía de clases se llaman sus descendientes (c_{21}, c_{22} y c_{23} son descendientes de c_2).
A medida que crece el número de descendientes se incrementa la reutilización para la abstracción representada por la clase predecesora puede verse diluida ya que puede ser que algunos descendientes no sean realmente miembros propios de ella.
Si NDD crece, la cantidad de pruebas necesarias para ejercitar cada descendiente en su contexto operativo también crecerá.
Acoplamiento entre clases objeto (ACO): es el número de colaboraciones enumeradas para una clase en su tarjeta índice CRC.
Si ACO crece es probable que baje la reutilización de la clase, se complican las modificaciones y la comprobación. ACO debiera mantenerse en los valores más bajos razonables.

Respuestas para una clase (RPC): el **conjunto de respuesta** de una clase es el conjunto de métodos que se pueden ser ejecutados potencialmente en respuesta a un mensaje recibidos por un objetos de la clase. RPC es el número de métodos existentes en el conjunto de respuestas.
A medida que RPC crece, crece también el esfuerzo necesario para la comprobación y la complejidad global de diseño de la clase crece también.

Carencia de cohesión en los métodos (CCM): se define como el número de métodos que acceden a uno o más de los mismos atributos. Si ningún método accede a los mismos atributos CCM es 0.
Por ejemplo, si en una clase de 6 métodos, cuatro (4) de los métodos tienen en común uno o más atributos, CCM=4.
Si CCM es alto, los métodos pueden estar acoplados entre sí a través de atributos, lo que incrementa la complejidad del diseño de clases. En general, valores elevados para CCM implican que la clase puede descomponerse en dos o más clases distintas. Aunque existen casos en que se justifica un valor elevado de CCM, es deseable mantener un alto grado de cohesión, es decir, un valor bajo de CCM.

18.3 Métricas de Lorenz y Kidd

Proponen dividir las métricas basadas en clases en 4 grandes categorías: **tamaño, herencia, valores internos y valores externos.**
Las métricas para una clase OO basadas en:

- **Tamaño:** se centran en recuentos de atributos y operaciones para una clase individual y promedian los valores para el sistema OO en su totalidad.
- **Herencia:** se centran en la forma en que se reutilizan las operaciones a lo largo y ancho de la jerarquía de clases.
- **Valores internos:** examinan la cohesión y asuntos relacionados con el código.
- **Valores externos:** examinan el acoplamiento y la reutilización.

Algunas de estas métricas son las siguientes:

Tamaño de clase (TC): el tamaño de una clase puede determinarse por:

- **El número total de operaciones (tanto heredadas como privadas) encapsuladas dentro de la clase.**
- **El número de atributos (tanto heredados como privados de la instancia).**

Los valores grandes de TC indican que una clase puede tener demasiada responsabilidad que reducirá la reutilización de la clase y complicará la implementación y la comprobación.

Para determinar el tamaño de una clase se ponderan con mayor importancia las operaciones públicas o heredadas y los atributos. Los atributos y operaciones privados hacen posible la especialización y están más localizadas en el diseño.

También se pueden calcular los promedios de atributos de la clase y operaciones. Cuanto menos sea el valor medio para el tamaño es más probable que las clases del sistema puedan reutilizar ampliamente.

Número de operaciones invalidadas por una subclase (NOI): se llama **invalidación** al caso en que una **subclase** sustituye una operación heredada de su superclase por una versión especializada para su propio uso.

Si NOI tiene un valor grande suele indicar un problema de diseño ya que el diseñador ha violado la abstracción implicada por la superclase donde la subclase (especialización de su superclase) debiera limitarse a extender los servicios de la superclase y no dar nuevos nombres a métodos únicos.

Esto da lugar a una jerarquía de clases débil y a un software OO difícil de comprobar y modificar.

Número de operaciones añadidas por una subclase (NOA): las subclases se especializan con la adición de operaciones y atributos privados. A medida que NOA crece, la subclase deriva con respecto a la abstracción implicada por la superclase. En general, cuando crece la profundidad de la jerarquía de clases, el valor de NOA en los niveles inferiores de la jerarquía debería disminuir.

Índice de especialización (IE): proporciona una indicación aproximada del grado de especialización de cada una de las subclases existentes en un sistema OO.
La especialización se puede alcanzar agregando o añadiendo operaciones, o bien por invalidación.

$$IE = [NOI \times nivel] M_{total}$$

Donde,

nivel — es el nivel de la jerarquía de clases en que reside la clase.

M_{total} — es el número total de métodos para la clase.

Cuanto más alto sea el valor de IE es más probable que la jerarquía de clases tenga clases que no se ajusten a la abstracción de la superclase.

18.4 Métricas Orientadas a Operaciones

Se han propuesto menos métricas para las operaciones de clases que para las clases ya que éstas últimas son la unidad dominante en los sistemas OO y que los métodos tienden a ser pequeños (en términos de tamaño y complejidad) e interesa más la estructura de conectividad de un sistema que el contenido de los módulos individuales.

Tamaño medio de operación (Toavg): aunque se podría usar la cantidad de líneas de código como indicador del tamaño de operación, se usa alternativamente el número de mensajes enviados por la operación como medida del tamaño de la operación. Si este número crece, es probable que las responsabilidades no hayan sido bien asignadas dentro de la clase.

Complejidad de operación (CO): se calcula la complejidad de una operación y se considera que ya que las operaciones debieran limitarse a una responsabilidad específica, el valor de CO debiera mantenerse tan bajo como sea posible.

Número medio de parámetros por operación (Npavg): a mayor número de parámetros de la operación, será más compleja la colaboración entre

objetos. En general, Npavg debería tratar de mantenerse tan bajo como sea posible.

18.5 Métricas para pruebas OO

Las métricas vistas anteriormente proporcionan una indicación de la calidad del diseño y de la cantidad de esfuerzo de pruebas necesarios para aplicarlo en un sistema OO.

Binder (94) sugiere una amplia gama de métricas de diseño que tiene influencia directa en la **comprobabilidad** de un sistema.
Estas métricas se organizan en categorías definidas en términos de características de diseño importantes.

18.5.1 Encapsulamiento

Carencia de cohesión en métodos (CCM): a más el valor de CCM, más estados habrá que probar para asegurar que los métodos no dan lugar a efectos laterales.

Porcentaje público y protegido (PPP): los atributos públicos se heredan de otras clases y ∴ son visibles para esas clases.
Los atributos protegidos son una especialización y son privados de una subclase específica.
PPP indica el porcentaje de atributos de clase que son públicos. Si PPP es un valor alto existe mayor probabilidad de efectos laterales entre clases y se necesita diseñar comprobaciones que aseguren que se descubran estos efectos laterales.

Acceso público a datos miembros (APD): esta métrica indica el número de clases (o métodos), que pueden acceder a los atributos de otra clase, violando el encapsulamiento.
A valores de APD hay más probabilidad de efectos laterales clases y es necesario diseñar comprobaciones que aseguren que estos efectos laterales se descubran.

18.5.2 Herencia

Número de clases raíz (NCR): es un recuento de las jerarquías de clase distintas que se describen en el modelo de diseño como es preciso desarrollar conjuntos de pruebas para cada clase raíz y su correspondiente jerarquía de clases, si NCR crece también crece el esfuerzo de comprobación.

Admisión (ADM): en el contexto OO, la admisión es una indicación de herencia múltiple. ADM >1 indica que una clase hereda sus atributos y

operaciones de más de una clase raíz. Siempre que sea posible, se debe evitar el valor de ADM >1.

Número de descendentes (NDD) y profundidad del árbol de herencia (APM): los métodos de la superclase tendrán que ser comprobados de nuevo para cada una de las subclases.

Binder también define métricas para la complejidad de la clase y para el polimorfismo. Para la complejidad incluye tres métricas ck: MPC, ACO y RPC. Se definen también métricas asociadas al recuento de métodos. Las métricas asociadas al polimorfismo son muy especializadas y su descripción no se realizará en este curso.

18.6 Métricas para proyectos OO

Ya que la tarea de un administrador de proyectos es planificar, coordinar, seguir y controlar un proyecto de software, interesa conocer algunas métricas que aporten ideas acerca del tamaño implementado del software durante la etapa de estimación en la planificación.

Número de guiones de escenarios (NGE): el número de guiones de escenario o casos prácticos es directamente proporcional al número de clases necesarias para satisfacer los requisitos, al número de estados de cada clase y al número de métodos, atributos y colaboraciones. NGE es un excelente indicador del tamaño del programa.

Número de clases clave (NCC): las clases clave se centran directamente en el dominio del negocio para el problema en cuestión y tendrán menor probabilidad de ser implementadas mediante reutilización. Así, valores altos de NCC indican que será necesario un gran trabajo de desarrollo. Se sugiere que entre el 20% y el 40% de todas las clases de un sistema OO típico son clave mientras que el resto sirve como infraestructura de apoyo.

Número de subsistemas (NSUB): el número de subsistemas proporciona una idea general de la asignación de recursos, de la planificación y del esfuerzo global de integración.

Estas métricas, junto con otras métricas de diseño, se pueden usar para calcular **métricas de productividad** tales como el número medio de clases por desarrollador o el promedio de métodos por persona y por mes. En conjunto, estas métricas sirven para estimar el esfuerzo, la duración, el personal y otra información para el proyecto actual.

GESTIÓN DE LA CALIDAD DEL SOFTWARE

- Ingeniería de Software
Ian Sommerville – 6ta Edición
- Calidad del Software
Juan Manuel Cueva Lovelle
Dpto. De Informática
Universidad de Oviedo – España
Ingeniería de Software – Un enfoque práctico
Roger Pressman – 6ta. Edición

GCS.1 Introducción

El objetivo principal de muchas organizaciones actuales es lograr un alto nivel de calidad de sus productos o servicios. Ya no es aceptable entregar productos de calidad pobre y arreglar los problemas o deficiencias después de la entrega al cliente. De la misma manera, el software tiene las mismas características que cualquier otro producto manufacturado.

Sin embargo la calidad del software es un concepto complejo que no se puede definir en forma simple. Clásicamente,

- **La noción de calidad es que el producto desarrollado cumple su especificación**. (Crosby, 1979)
- Calidad es **"la concordancia con los requisitos funcionales y de rendimiento establecidos con los estándares de desarrollo debidamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente"**. (R. Pressman, 1992)
- "**El conjunto de características de una entidad que le confieren su aptitud para satisfacer las necesidades expresadas y las implícitas**" (ISO 8402, UNE 66-001-92).

En un mundo ideal, estas definiciones aplican a todos los productos, pero, para sistemas de software, existen problemas:

1. La especificación se orienta hacia las características del producto que el consumidor quiere. Sin embargo, la organización desarrolladora también tiene requerimientos (ej. los de mantenimiento) que no se incluyen en la especificación.

2. No se sabe cómo especificar ciertas características de calidad (por ejemplo, mantenimiento) de una forma no ambigua.

3. Es muy difícil redactar especificaciones concretas de software. Por lo tanto, aunque un producto esté acorde a su especificación, los usuarios no lo consideran un producto de alta calidad.

Aunque se hagan esfuerzos para mejorar las especificaciones, actualmente se tiene que aceptar que son imperfectas. Dentro de este marco, se diseñan procedimientos para mejorar la calidad. En particular, los atributos del software, como la mantenibilidad, la portabilidad, o la eficiencia, son atributos de calidad críticos que no se especifican explícitamente, pero que afectan la calidad percibida del sistema.

Ya que, los requisitos del software son la base de las medidas de calidad y la falta de concordancia con los requisitos es una *falta de calidad*, la responsabilidad de los administradores de la calidad en una organización es asegurar que se cumpla el nivel requerido de la calidad del producto definiendo estándares o metodologías y procedimientos durante el desarrollo de software y comprobando que todos los ingenieros los sigan. Si no se sigue ninguna metodología, siempre habrá falta de calidad.

Sin embargo, la gestión de calidad es más que esto. Los buenos administradores de calidad tienen como propósito desarrollar una "cultura de calidad", donde cada participante del proyecto es motivado para que logre un alto nivel del calidad del producto. Se fomento que los equipos tomen responsabilidad de la calidad de su trabajo y desarrollen nuevos enfoques de mejora de la calidad. Aunque seguir los estándares y procedimientos es la base de la gestión de calidad, existen aspectos intangibles para la calidad del software (elegancia, trasparencia, etc.) que no están incluidos en los estándares. El administrador de calidad del software debe apoyar al personal interesado en estos aspectos y fomentar el comportamiento profesional de todos los miembros del equipo.

GCS. 2 Proceso de Gestión de la calidad

El proceso de gestión de la calidad del software se estructura en tres actividades principales:

1. ***Aseguramiento de la calidad***: Es el establecimiento de un marco de trabajo de procedimientos y estándares organizacionales que conduce a software de alta calidad.
2. ***Planificación de la calidad***: Es la selección de procedimientos y estándares adecuados a partir de este marco de trabajo y la adaptación de éstos para un proyecto de software específico.
3. ***Control de la calidad***: La definición y promulgación de los procesos que aseguren que los procedimientos y estándares para la calidad del proyecto son seguidos por el equipo de desarrollo de software.

La definición de Gestión de calidad (ISO 9000):

> **Conjunto de actividades de la función general de la dirección que determina la calidad, los objetivos y las responsabilidades, y se implementa por medios tales como la planificación de la calidad, el control de la calidad, el aseguramiento (garantía) de calidad y la mejora de la calidad, el marco del sistema de calidad.**

También ISO-9000, define como política de calidad a "Directrices y objetivos generales de una organización, relativos a la calidad, tal como se expresan formalmente por alta dirección".

La gestión de calidad se aplica, normalmente a nivel de empresa, aunque puede haber gestión de la calidad dentro de la gestión de cada proyecto.

GCS.2.1 Aseguramiento de la calidad

El aseguramiento de la calidad del software es el ***conjunto de actividades planificadas y sistemáticas necesarias para aportar la confianza en que producto (software) satisfará los requisitos dados de calidad***.

El aseguramiento de calidad del software se diseña para cada aplicación ***antes de comenzar a desarrollarla*** y no después.

Algunos autores prefieren decir **garantía** de calidad en lugar de aseguramiento, pero la palabra garantía se puede confundir con la garantía de productos, y la palabra aseguramiento pretende dar confianza en que el producto tiene calidad.

Se asegura la calidad cuando:

- Se seleccionan métodos y herramientas de análisis, diseño, programación y prueba.

- Se realizan inspecciones técnicas formales en todos los pasos del proceso de desarrollo de software.

- Se definen estrategias de prueba multiescala.

- Se hacen controles de la documentación del software y de los cambios realizados.

- Se definen procedimientos para ajustarse a los estándares (y se deja claro cuándo se está fuera de ellos).

- Se definen mecanismos de medida (métricas).

- Se registran auditorías e informes.

Las actividades previstas para el aseguramiento de la calidad son:

- Métricas del software para el control del proyecto.
- Verificación y validación del software a lo largo del ciclo de vida.
  Incluye las pruebas y los procesos de revisión e inspección.
- La gestión de la configuración del software.

Existen dos tipos de estándares que se establecen como parte del proceso de aseguramiento de la calidad.

1. ***Estándares del producto***: Son los estándares que aplican al producto de software a desarrollar. Incluyen estándares de documentos (ej. La estructura del documento de requerimientos a producir), estándares de documentación (ej. Encabezados estándar de comentarios para definiciones de clases y objetos) y estándares de codificación (definen cómo usar un lenguaje de programación).
2. ***Estándares del proceso***: Son los estándares que definen los procesos a seguir durante el desarrollo de software. Incluyen definiciones de procesos de especificación, de diseño y de validación y una descripción de los documentos a generar en el transcurso de estos procesos.

Existe una relación cercana entre los estándares del producto y del proceso. Los estándares del producto se aplican a las salidas del proceso de software y, en muchos casos, los estándares del proceso incluyen actividades específicas del proceso que aseguran que se siguen los estándares del producto.

Los estándares de software son importantes porque:

a) Proveen un conjunto compacto de las mejores prácticas, o al menos las apropiadas. Este conocimiento se adquiere, a menudo, luego de seguir un proceso de prueba y error. Tenerlo incorporado en un estándar evita la repetición de errores. **Los estándares capturan conocimiento de valor para la organización**.
b) Proveen un marco de trabajo alrededor del cual se implementa el proceso de mejoramiento de la calidad. Ya que los estándares capturan las mejores prácticas, **el control de la calidad simplemente asegura que los estándares se siguen adecuadamente**.
c) Ayudan a la continuidad cuando una persona lleva a cabo el trabajo otra lo continúa. Los estándares aseguran que todos los ingenieros dentro de una organización adopten las mismas prácticas. En consecuencia, **se reduce el esfuerzo de aprendizaje cuando se comienza un nuevo trabajo**.

Los estándares de documentación en un proyecto de software son muy importantes ya que son la única forma tangible de representar el software y el proceso del software. Los documentos estandarizados tienen una apariencia, estructura y calidad consistentes, y por lo tanto son fáciles de leer y comprender.

Existen tres tipos de estándares de documentación:

i) ***Estándares del proceso de documentación***: Que definen el proceso a seguir para la producción del documento.
ii) ***Estándares del documento***: Que gobiernan la estructura y presentación de los documentos.
iii) ***Estándares para el intercambio de documentos***: Que aseguran que todas las copias electrónicas de los documentos sean compatibles.

Algunos ejemplos de estándares de estos tipos son: estándares para la identificación de documentos (cada documento debe identificarse de forma única con un identificador formal definido por el administrador); estándares de la estructura del documento (Cada documento debe seguir una estructura estándar que defina secciones, numeración de páginas, encabezados, etc.); estándares de presentación de documentos (definición de tipos de letras y estilos, logotipos y nombres de la compañía, uso de colores, etc.) y estándares para actualizar documentos (Los cambios en los documentos se deben hacer de forma consistente, usando colores para resaltar las versiones, indicar los párrafos modificados o agregados, etc.).

GCS.2.2 Planificación de la Calidad

Un plan de calidad define la calidad del producto deseado y cómo valorarla. Esta parte del proceso comienza en las primeras etapas del proceso del software.

Un plan de calidad define lo que significa software de "alta calidad" y es el resultado de la planificación de calidad de un proyecto. Este plan de calidad selecciona los estándares organizacionales apropiados para un producto particular y un proceso de desarrollo. Si el proyecto usa nuevos métodos y herramientas se deben definir nuevos estándares de calidad.

La estructura sugerida por Humphrey (1989) para un plan de calidad comprende:

i) ***Introducción del producto***: Descripción del producto, el mercado al que se dirige y expectativas de calidad del producto.
ii) ***Planes del producto***: Contiene las fechas de terminación del producto y las responsabilidades importantes, junto con los planes para distribución y servicio.
iii) ***Descripciones del proceso***: Contiene procesos de desarrollo y de servicio a usar para el desarrollo y gestión del producto.
iv) ***Metas de calidad***: Contiene las metas y planes de calidad para el producto, incluye una identificación y justificación de los atributos de calidad importantes del producto.

v) ***Riesgos y gestión de riesgos***: Contiene los riesgos claves que podrían afectar la calidad del producto y las acciones para abordar estos riesgos.

Los planes de calidad deben ser redactados de la forma más compacta posible de modo que los ingenieros los lean y se ajusten al plan.

Algunos de los atributos de calidad potenciales del software son:

Seguridad	Comprensión	Portabilidad
Protección	Experimentación	Usabilidad
Fiabilidad	Adaptabilidad	Reutilización
Flexibilidad	Modularidad	Eficiencia
Robustez	Complejidad	Aprendizaje

En general, no es posible optimizar todos estos atributos para un sistema. Una parte importante de la planificación de la calidad es seleccionar los atributos de calidad importantes y planear cómo alcanzarlos.

Un plan de calidad define los atributos de calidad más importantes para el producto a desarrollar, aunque debe sacrificar otros atributos para alcanzarla. El plan también define el proceso de valoración de la calidad.

En resumen:

• Un plan de calidad consta de una estructura organizativa, procedimientos, procesos y recursos necesarios para implementar la gestión de la calidad.

• Un sistema de calidad se debe adecuar a los objetivos de calidad de la empresa.

• La dirección de la empresa es la responsable de fijar la política de calidad y las decisiones relativas a iniciar, desarrollar, implementar y actualizar el sistema de calidad.

• Un sistema de calidad consta de varias partes:

i) ***Documentación***: Manual de calidad. Es el documento principal para establecer e implementar un sistema de calidad. Puede haber manuales a nivel de empresa, departamento, producto, específicos (compras, proyectos, etc.).

ii) ***Parte Física***: Locales, herramientas, computadoras, etc.

iii) ***Aspectos humanos***: Formación de personal y creación y coordinación de equipos de trabajo.

GCS.2.2.1 Certificación de la Calidad

Un sistema de certificación de la calidad permite una valoración independiente que debe demostrar que la organización es capaz de desarrollar productos y servicios de calidad.

Los pilares básicos de la certificación de calidad son tres:

1) Una metodología adecuada.
2) Un medio de valoración de la metodología
3) Tanto la metodología utilizada como el medio de valoración de la metodología deben estar ampliamente reconocidos por la industria.

GCS.2.3 Control de la calidad

Implica vigilar el preso de desarrollo de software para asegurar que se sigan los procedimientos de aseguramiento y estándares de calidad. En otras palabras, son las actividades para evaluar la calidad de los productos desarrollados.

> **Control de calidad son las técnicas y actividades de carácter operativo, utilizadas para satisfacer los requisitos relativos a la calidad, centradas en dos objetivos fundamentales:**
>
> **i) Mantener bajo control el proceso.**
> **ii) Eliminar las causas de los defectos en las diferentes fases del ciclo de vida.**

El proceso de control de calidad tiene su propio conjunto de procedimientos e informes a usar durante el proceso de desarrollo. Estos son directos y fácilmente comprensibles por ingenieros que desarrollan software.

Existen dos enfoques complementarios para el control de la calidad:

1) ***Revisiones de calidad*** en las que el software, su documentación y los procesos utilizados para producir software son revisados por un grupo de personas que son los responsables de comprobar que se han seguido los estándares del proyecto y los documentos están acordes a ellos. Toman las desviaciones de los estándares y las ponen a consideración de la administración del proyecto.
2) ***Valoración automática del software*** en la que el mismo y los documentos se procesan por algún programa y se comparan con los estándares que aplican al presente proyecto de desarrollo. Esta valoración automática comprende una medida cuantitativa de algunos atributos del software.

Las revisiones de calidad son el método más usado para validar la calidad de un proceso o producto. Involucran a un grupo de personas que examinan parte o todo el proceso del software, los sistemas o su documentación asociada para descubrir problemas potenciales. Las conclusiones de la revisión se registran formalmente y se pasan a quien es responsable de corregir los errores descubiertos.

Algunos tipos de revisiones de de calidad son:

a) ***Inspecciones de diseño o programas***: Su objetivo es detectar errores finos en los requerimientos, el diseño o el código. Se conduce por una lista de verificación de los posibles errores.

b) ***Revisiones de progreso***: Su objetivo es proveer información para administrar el progreso completo del proyecto. Es una revisión tanto del proceso como del producto y se refiere a los costos, planes y calendarización.

c) ***Revisiones de calidad***: Su objetivo es llevar a cabo un análisis técnico de los componentes del producto o documentación para encontrar diferencias entre la especificación y el diseño del componente, código y documentación y para asegurar que se siguen los estándares de calidad definidos.

GCS.3 Métricas de calidad del software

Se puede medir la calidad a lo largo del proceso de Ingeniería de Software y una vez que el software ha sido distribuido al cliente y a los usuarios.

• Las **métricas obtenidas antes de entregar el software** proporcionan una base cuantitativa sobre la que tomar decisiones de diseño y prueba. Algunos ejemplos de estas métricas son: ***complejidad del programa***, ***modularidad efectiva*** y el ***tamaño global del programa***.

• Las **métricas que se usan después de la distribución** se centran en el ***número de defectos no descubiertos durante la prueba*** y en la ***facilidad de mantenimiento***.

Es importante señalar que las medidas post-distribución de la calidad del software suponen para los gestores y los técnicos una indicación a posteriori del la efectividad del proceso de ingeniería de software.

Hace más de 20 años se definió un conjunto de factores de calidad que constituyeron el primer paso hacia el desarrollo de métricas para la calidad del software.

Se clasifican en tres grupos:

A) **La operación del producto (Su uso)**: Son características operativas:
   - **i)** **Corrección** – El grado en que una aplicación satisface sus especificaciones y consigue los objetivos encomendados por el cliente (Hace lo que se le pide?).
   - **ii)** **Fiabilidad**: El grado que se puede esperar de una aplicación para que lleve a cabo las operaciones

especificadas y con la precisión requerida. (Lo hace de forma fiable todo el tiempo?).

iii) **Eficiencia**: La cantidad de recursos de hardware y software que necesita una aplicación para realizar las operaciones con los tiempos de respuesta adecuados. (Qué recursos de hardware y software necesito?).

iv) **Integridad**: El grado con que puede controlarse el acceso al software o los datos a personal no autorizado. (Puedo controlar su uso?).

v) **Facilidad de uso**: El esfuerzo requerido para aprender el manejo de una aplicación, trabajar con ella, introducir datos y conseguir resultados. (Es fácil y cómodo de manejar?).

B) **La revisión del producto (su modificación)**: Es la capacidad para soportar cambios:
   - i) **Facilidad de mantenimiento**: El esfuerzo requerido para localizar y reparar errores. (Puedo localizar los fallos?).
   - ii) **Flexibilidad**: El esfuerzo requerido para modificar una aplicación en funcionamiento. (Puedo añadir nuevas opciones?).
   - iii) **Facilidad de prueba**: El esfuerzo requerido para probar una aplicación de forma que cumpla con lo especificado en los requisitos. (Puedo probar todas las opciones?).

C) **La transición del producto(su transportabilidad)**: Es la adaptabilidad a nuevos entornos:
   - i) **Portabilidad**: El esfuerzo requerido para transferir la aplicación a otro hardware o sistema operativo. (Podré usarlo en otra máquina?).
   - ii) **Reusabilidad**: Grado en que partes de una aplicación puede usarse en otras aplicaciones. (Podré utilizar alguna parte del software en otra aplicación?).
   - iii) **Interoperabilidad**: El esfuerzo necesario para comunicar la aplicación con otras aplicaciones o sistemas informáticos. (Podrá comunicarse con otras aplicaciones o sistemas informáticos?).

Es difícil, y en algunos casos imposible, desarrollar medidas directas de los factores de calidad del software.

Cada factor de calidad, F_c se puede obtener como una combinación de de una o varias métricas:

$$F_c = c_1 * m_1 + c_2 * m_2 + ... + c_n * m_n$$

Donde:

c_i es una factor de ponderación de la métrica i, que dependerá de cada aplicación específica.
m_i es la métrica i

Habitualmente se puntúan entre 0 y 10 las métricas y los factores de calidad.

Algunas métricas para determinar factores de calidad son:

Facilidad de auditoría	Exactitud
Normalización de las comunicaciones	Completitud
Concisión	Consistencia
Estandarización de los datos	Tolerancia de errores
Eficiencia de ejecución	Facilidad de expansión
Generalidad	Independencia del hardware
Instrumentación	Modularidad
Facilidad de operación	Seguridad
Autodocumentación	Simplicidad
Independencia del sistema de software	
Facilidad de traza	Formación

GCS.3.1 Medida de calidad

Aunque existen muchas formas de medir la calidad del software, las métricas a posteriori son las más ampliamente usadas. Estas incluyen **facilidad de mantenimiento**, **integridad** y **facilidad de uso**.

Se sugieren las siguientes definiciones y medidas para cada una de ellas:

Corrección o Correctitud: Un sistema tiene que funcionar correctamente para que tenga valor para el usuario. ***La corrección es el grado con que el software realiza la función requerida***. La medida más común es el número de defectos por KLOC

Corrección = nro. de defectos / KLOC

Donde:

Defecto: Se define como ***carencia verificada de conformidad con los requisitos***. Se cuentan durante un período de tiempo estándar, típicamente, un año. Los comunica el usuario del sistema después de que el mismo ha sido distribuido para su uso general.

Facilidad de mantenimiento: El mantenimiento de software es la actividad de la Ingeniería de Software que más esfuerzo representa. Se define como ***la facilidad con que se puede corregir un programa si se encuentra un error, adaptarlo si su entorno cambia o mejorarlo si el cliente desea un cambio en los requisitos***.

Como no hay forma de medir directamente la facilidad mantenimiento, se usan medidas indirectas. Una métrica sencilla orientada al tiempo es:

TMEC = Tiempo medio entre cambios

Es decir, el tiempo que lleva analizar el cambio requerido, diseñar una modificación apropiada, implementar el cambio, probarlo y distribuirlo a todos los usuarios.

Naturalmente, los programas fáciles de mantener tendrán, para cambios equivalentes, un menor TMEC que los programas que no son fáciles de mantener.

Integridad: En esta época de piratas y virus, la integridad del software ha ganado cada vez más importancia. Este atributo ***mide la habilidad de un sistema a resistir ataques, tanto accidentales como intencionados, contra su seguridad***.

Se define como:

Amenaza: La probabilidad de que un ataque determinado ocurra en un tiempo determinado.

Seguridad: La probabilidad de que se pueda repeler un ataque de un determinado tipo.

Entonces, la integridad del sistema se define como:

Integridad = Σ[1 – amenaza * (1 – seguridad)]

Donde se suman la amenaza y la seguridad para cada tipo de ataque.

Facilidad de uso: El calificativo "amigable" con el usuario se ha convertido en omnipresente en las discusiones sobre productos de software. Tanto es así que, si un producto no es amigable con el usuario, generalmente, va destinado al fracaso. La facilidad de uso es un intento de cuantificar esa "amigabilidad" con el usuario y se puede medir en función de cuatro (4) características:

a) Habilidad intelectual o física para aprender el sistema.
b) Tiempo requerido para llegar a ser moderadamente hábil en el uso del sistema.
c) Aumento neto en la productividad, medida cuando al sistema lo usa alguien moderadamente eficiente.
d) Valoración subjetiva de la disposición de los usuarios al sistema.

GCS.3.2 Reconciliación de diferentes métricas

La siguiente lista proporciona estimaciones informales del número medio de líneas de código para construir un punto de función en alguno lenguajes de programación:

Ensamblador	300
COBOL	100
FORTRAN	100
Pascal	90
Ada	70
Lenguajes Orientados a Objetos	30
Lenguajes de 4ta Generación (L4G)	20
Generadores de Código	15

Esta tabla muestra, por ejemplo, que una línea de código en Ada proporciona aproximadamente 1,4 veces más funcionalidad (media) que una línea de código de FORTRAN.

Cualquier discusión sobre medidas de productividad del software, irremisiblemente termina en un debate acerca del uso de los datos. ¿Se debe comparar la relación LOC/personas-mes (o PF/personas-mes) de un grupo con los datos similares de otro grupo? ¿Deben los gestores evaluar el rendimiento

de las personas usando estas medidas? La respuesta a estas preguntas es un rotundo NO, ya que hay muchos factores que influyen en la productividad haciendo que la comparación de "peras con manzanas" sea fácilmente mal interpretada.

Hay cinco **(5) factores importantes que inciden en la productividad** del software:

Factores Humanos: El tamaño ya la experiencia de la organización de desarrollo.

Factores del problema: La complejidad del problema a resolver y el número de cambios en las restricciones o los requisitos del diseño.

Factores del proceso: Técnicas de análisis y diseño usadas, disponibilidad de lenguajes y de herramientas CASE y técnicas de revisión.

Factores del producto: Fiabilidad y rendimiento del sistema basado en computadoras.

Factores de los recursos: Disponibilidad de herramientas CASE, de recursos De Hardware y Software.

Se han dirigido unas pruebas estándar para ilustrar el efecto de estos factores. Si alguno de ellos está por encima de la media (altamente favorable) para un determinado proyecto, la productividad de desarrollo de software será significativamente más alta que si el mismo factor estuviera por debajo de la media. Para los cinco factores indicados arriba, los cambios desde condiciones altamente favorables a condiciones desfavorables afectarán la productividad de la siguiente manera:

Factores Humanos	90% de variación
Factores del problema	40% de variación
Factores del proceso	50% de variación
Factores del producto	140% de variación
Factores de los recursos	40% de variación

En otras palabras, si dos equipos de ingeniería de software tienen gente con las mismas habilidades, usando los mismos recursos en el mismo proceso, uno de los equipos trabajando con un problema relativamente simple, con requisitos de rendimiento y fiabilidad medios, y el otro equipos está trabajando en un problema complejo, con objetivos de fiabilidad y rendimiento extremadamente altos, entonces, basándose en la tabla anterior, el primer equipo podrá exhibir una productividad de desarrollo de software que esté entre un 40%y un 140% mejor que el segundo equipo. Es entonces que se ve que los factores del problema y del producto hacen que la comparación de la productividad entre los dos equipos no tenga sentido.

GCS.3.3 Integración de las métricas dentro del proceso de Ingeniería de Software

Si no medimos el proceso de ingeniería de software, a pesar de las resistencias que ponen los desarrolladores a hacerlo, no hay forma de determinar si estamos mejorando. ***La medición proporciona beneficios a nivel estratégico, a nivel de proyecto y a nivel técnico***.

Mediante la obtención y evaluación de las medidas de calidad y de productividad, los directivos de gestión pueden establecer objetivos significativos para mejorar el proceso de Ingeniería de Software. Ya que el software es un objetivo comercial estratégico para muchas empresas, si se puede mejorar el proceso a través del cual se desarrolla, se puede producir un impacto directo en lo sustancial. La medición se usa para establecer una línea base del proceso a partir de la cual se pueden evaluar mejoras.

A nivel técnico, las métricas del software, cuando se aplican al producto, **proporcionan beneficios inmediatos.** Cuando se ha terminado el diseño del software crece la ansiedad de responder a preguntas tales como:

¿Qué requisitos del usuario son más susceptibles al cambio?
¿Qué módulos del sistema son más propensos a errores?
¿Cómo se debe planificar una prueba para cada módulo?
¿Cuántos errores de cada tipo se pueden esperar cuando comience la prueba?

El haber recopilado métricas y se han usado como guía técnica se pueden encontrar estas respuestas.

Establecimiento de una línea de base

A pesar de todo, la información recogida no ha de ser esencialmente diferente para satisfacer cada una de las diferentes circunscripciones descriptas anteriormente. La forma en que se presente la información será diferente, pero las mismas métricas pueden ser útiles para muchos gestores.

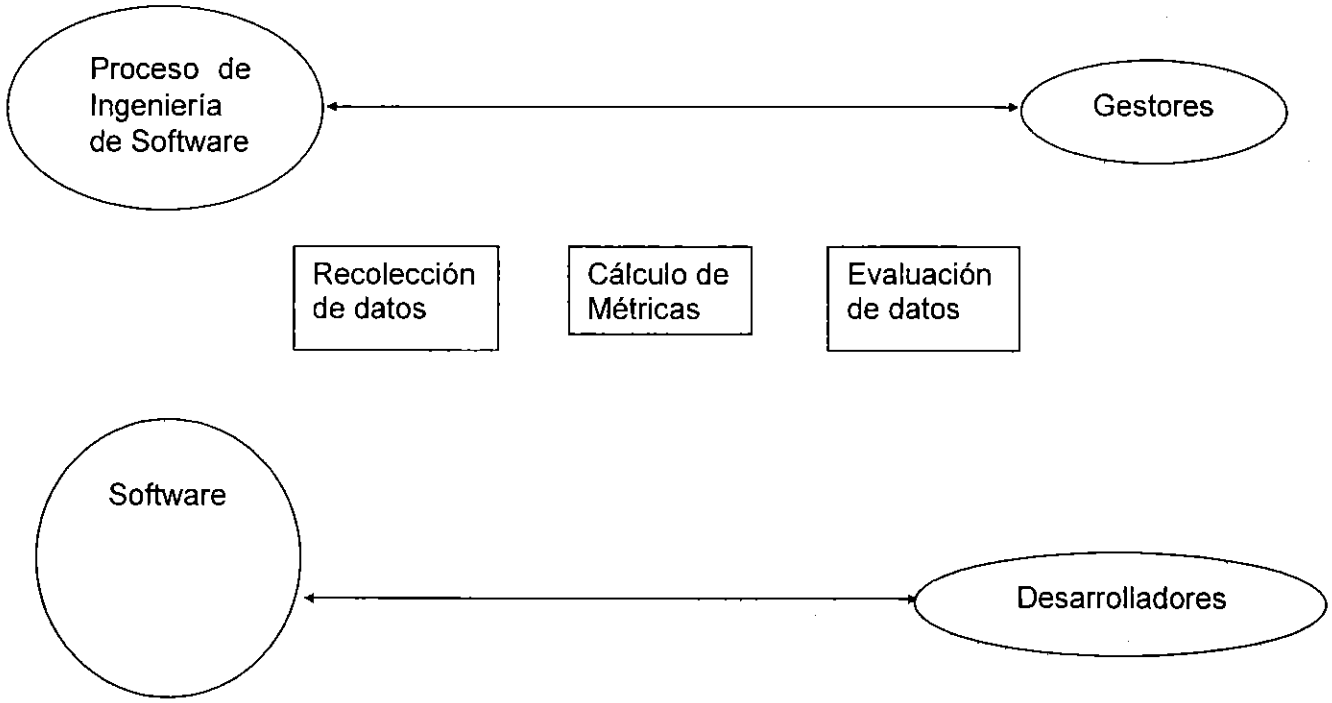
La **línea base** consiste en datos recogidos de anteriores proyectos de desarrollo de software y puede ser muy sencilla o muy compleja. Además de simples medidas orientadas al tamaño o a la función, la línea base se puede complementar con métricas de calidad.

Para que sea una ayuda efectiva en la planificación estratégica y/o en las estructuraciones de coste y esfuerzos, los datos de la línea base tienen que poseer los siguientes atributos:

(1) Los datos deben ser razonablemente precisos, evitando las suposiciones sobre proyectos anteriores.
(2) Los datos deben obtenerse de tantos proyectos como sea posible.
(3) Las medidas tienen que ser consistentes (por ejemplo, las LOC deben interpretarse de la misma forma para todos los proyectos)
(4) Las aplicaciones deben ser similares a la que vaya a ser estimada.

Recolección, cálculo y evaluación de métricas

La siguiente figura ilustra el proceso que se sigue para establecer una línea base:



Idealmente, los datos necesarios para establecer la línea base han sido obtenidos de forma directa, aunque raramente esto es cierto. Generalmente, la ***recolección de datos*** requiere una investigación histórica de proyectos pasados para reconstruir los datos requeridos.

Una vez que se han obtenido los datos, se pasa al ***cálculo de métricas*** que pueden abarcar un amplio rango de medidas de LDC o PF de acuerdo a la amplitud de los datos obtenidos.

Por último, los datos calculados tienen que evaluarse y aplicarse en la estimación. La ***evaluación de datos*** se centra en las razones intrínsecas de los resultados obtenidos.

Conclusión

La gestión del proyecto de software representa el primer nivel del proceso de Ingeniería de Software. Está compuesta por actividades que incluyen medición, estimación, análisis de riesgos, planificación, seguimiento y control.

La medición permite a los gestores y desarrolladores entender mejor el proceso de Ingeniería de Software y el producto que se produce. Las métricas para la productividad y la calidad se pueden definir mediante medidas directas e indirectas

En la industria se usan tanto las métricas orientadas al tamaño como las orientadas a la función.

La Gestión Del Proyecto: Estimación
Introducción

El proceso de gestión del proyecto de software comienza con un conjunto de actividades que, globalmente, se denominan ***planificación*** del proyecto. La primera de estas actividades se llama ***estimación.***

Aunque la estimación es más un arte que una ciencia, es una actividad importante que no debe llevarse a cabo en forma descuidada. Existen técnicas útiles para la estimación de costos y de tiempos.

La estimación de recursos, costos y agendas para el esfuerzo de desarrollo de software requiere experiencia, acceso a una buena información histórica y coraje para confiar en soluciones cuantitativas cuando todo lo que existe son datos cualitativos.

La ***complejidad del proyecto*** tiene un gran efecto sobre la incertidumbre, que es inherente a la planificación., aunque es una medida relativa que se ve afectada por la familiaridad con anteriores esfuerzos.

El ***tamaño del proyecto*** es otro factor importante que puede afectar precisión y la eficacia de las estimaciones ya que a medida que aumenta tamaño, crece la interdependencia entre los distintos elementos del software.

El ***grado de estructuración*** del proyecto también tiene efecto sobre el riesgo de la estimación. En este contexto, la estructuración se refiere a que las funciones pueden ser compartamentalizadas y a la naturaleza jerárquica de la información que debe ser procesada.

La disponibilidad de información histórica también determina el riesgo de la estimación. Cuando se dispone de una amplia métrica del software de proyectos pasados se pueden evitar anteriores dificultades y se puede reducir el riesgo global.

El riesgo se mide por el grado de incertidumbre de las estimaciones cuantitativas establecidas para los recursos, los costos y las agendas. Si se vislumbra pobremente el proyecto o sus requisitos los mismos están sujetos a cambios, la incertidumbre y el riesgo llegan a ser peligrosamente altos. El planificador del software debe exigir información completa sobre la función, el rendimiento y las definiciones de la interfaz (contenida en la *especificación del sistema)* y tanto él como el cliente deben tener siempre presente que cualquier cambio en los requisitos del software significa inestabilidad en el costo y en la agenda.

Objetivos de la planificación del Proyecto

El objetivo de la planificación del proyecto de software es el de suministrar una estructura que permita al director hacer estimaciones razonables de recursos,

costos y agendas. Estas estimaciones se hace en un marco de tiempo limitado, al principio del proyecto de software y deben ser actualizadas regularmente a medida que progresa el proyecto.

Este objetivo se alcanza a través de un proceso de descubrimiento de información que lleve a estimaciones razonables.

A continuación veremos las actividades asociadas con la planificación del proyecto.

Ámbito del Software

Determinar el ***ámbito del software*** es la primera actividad de la planificación del proyecto. Se debe evaluar la función y el rendimiento asignados al software durante la ingeniería del sistema de computadoras con el fin de establecer un ámbito del proyecto que no sea ambiguo ni incomprensible a nivel de directivos y técnicos.

La especificación del ámbito del software debe estar delimitada, es decir:

a) Han de **establecerse explícitamente los datos cuantitativos** (número de usuarios simultáneos, tamaño de las listas de correo, tiempo máximo de respuesta posible).
b) **Han de señalarse las restricciones *y/o* limitaciones** (el costo del producto limita el tamaño de ¡a memoria)
c) **Han de describirse los factores de mitigación** (los algoritmos deseados se entienden bien y están disponibles en el lenguaje indicado)

El ámbito del software describe la función, las restricciones, las interfaces y la fiabilidad. Las **funciones** descriptas en la especificación del ámbito se evalúan y en algunos casos se refinan para dar más detalle antes de comienzo de las estimacion.

Las consideraciones de **rendimiento** abarcan los requisitos de tiempo de respuesta y de procesamiento.

Las **restricciones** identifican límites del software debidos al hardware externo, a ¡a memoria disponible y a otros sistemas existentes.

El software interactúa con otros elementos del sistema informático. El planificador considera la naturaleza y la complejidad de cada interfaz para determinar cualquier efecto sobre ¡os recursos, los costos y la agenda de desarrollo.

El **concepto de interfaz abarca:**

a) **Hardware** (procesador, periféricos) que ejecuta el software y dispositivos controlados indirectamente por el software (máquinas, pantallas)

b) El **software ya existente** (rutinas de acceso a base de datos, paquetes de rutinas, sistemas operativos).
c) **Gente** que hace uso del software por medio de terminales u otros medios de entrada/ salida.
d) **Procedimientos** que preceden o suceden al software en una secuencia de operaciones

El aspecto menos preciso del ámbito del software es el estudio de su **fiabilidad.** Aunque existen medidas del fiabilidad del software, rara vez se usan en esta etapa del proyecto. Las características de fiabilidad del hardware clásico, tales como el tiempo medio entre faltos, son difíciles de traducir al ámbito del software. Sin embargo, la naturaleza general del software puede dictar consideraciones especiales para asegurar la fiabilidad. Por ejemplo, algunos sistemas, en particular los que involucran a personas (control de tráfico aéreo) no pueden fallar. Otros no deberían fallar, pero el impacto de fallo es considerablemente menos dramático. Se puede usar la naturaleza del proyecto para la formulación de estimaciones de esfuerzo y de costo que aseguren la fiabilidad.

Si se ha formulado adecuadamente la **especificación del sistema** casi toda la información requerida para la descripción del ámbito del software estará disponible y documentada antes de que comience la planificación del proyecto.

Recursos

La segunda tarea de la planificación del desarrollo de software es la estimación de los recursos requeridos para acometer el esfuerzo de desarrollo del software.

En la siguiente gráfica se ilustran las recursos de desarrollo en. forma de pirámide:

Gente	**Especificar** • Habilidades requeridas • Disponibilidad • Duración de tareas • Fecha de comienzo
Herramientas Hardware/Software	**Especificar** • Descripción • Disponibilidad • Duración de uso • Fecha de distribución

En la base se encuentran las herramientas existentes - hardware y software - para dar soporte al esfuerzo de desarrollo. En el nivel más alto se encuentra el recurso primario que se requiere siempre - la gente. Cada recurso queda

especificado mediante cuatro características:

descripción del recurso
informe de disponibilidad
fecha cronológica en la que se lo requiere
tiempo durante el cual será aplicado

Recursos Humanos

El planificador comienza evaluando el ámbito y seleccionando las habilidades técnicas que se requieren para llevar a cabo el desarrollo. Hay que especificar, tanto la ***posición dentro de la organización*** (ej.: gestor, ingeniero senior, etc.) como la ***especialidad*** (ej.: telecomunicaciones, bases de datos, microprocesadores, etc.).Para proyectos relativamente pequeños, una persona puede llevar a cabo todos los pasos de la ingeniería de software, consultando con especialistas siempre que lo requiera.

El número de personas requeridas para un proyecto de software sólo puede ser determinado después de hacer una estimación del esfuerzo de desarrollo (ej.: personas-mes, personas-año).

Recursos de Hardware

El hardware es, dentro del ámbito de los recursos, no sólo el potencial de cálculo sino también una herramienta para el desarrollo del software.

Durante la planificación del proyecto se consideran tres categorías de hardware: el sistema de desarrollo, la máquina objetivo y los demás elementos de hardware del nuevo sistema.

- El **sistema de desarrollo** (o Sistema Anfitrión) está compuesto por la computadora que se usará durante la fase de desarrollo del software y sus periféricos asociados. (La máquina donde se ejecutará el software se denomina la **Máquina Objetivo**). Además se puede especificar como recursos para el desarrollo de software **otros elementos de hardware.**

Recursos de Software

Se usa el este software como ayuda en el desarrollo de nuevo software. La primera aplicación que se le dio al software fue lo que denominaba ***reconstrucción.*** Se comenzó escribiendo un primitivo traductor de lenguaje ensamblador a lenguaje de máquina y se usó para desarrollar un ensamblador más sofisticado. Aumentando las posibilidades de cada versión previa, los equipos de desarrollo fueron reconstruyendo eventualmente el software, hasta llegar a construir compiladores de lenguajes de alto nivel y otras herramientas.

Hoy, los ingenieros de software, usan un conjunto de herramientas parecidas,

en muchos casos, a las que usaban los ingenieros de hardware en el diseño y las ingeniería asistida por computadora (CAO/CAE).

Este conjunto de herramientas, **llamadas CASE**, se compone de las siguientes categorías:

Herramientas de Planificación de sistemas comerciales

Proporcionan una "meta modelo" a partir del cual se obtienen sistemas de información concretos mediante las modelización de los requisitos de información estratégica de una organización. Ayudan a crear sistemas de información que dirijan los datos a todos aquellos que necesiten la información e impidan el paso de la información no relevante hacia los miembros del personal.
Se optimiza la transferencia de datos y se llevan a cabo tomas de decisiones.

Herramientas de Gestión de Proyectos

Usando estas herramientas, el gestor puede generar estimaciones útiles de esfuerzo, costos y duración de los proyectos de software, puede definir una estructura de descomposición del trabajo, planificar una agenda factible y seguir la pista de los proyectos sobre una base uniforme.
Puede usar la herramienta para obtener métricas con las que establecer una línea base para la produ̇ctividad del proceso de desarrollo del software y para la calidad de los productos.

Herramientas de Soporte

Son las herramientas de producción de documentos, el software de sistemas en red, las bases de datos, el correo electrónico, los tablones de anuncios y las herramientas de gestión de configuraciones.

Herramientas de análisis y diseño

Permiten al ingeniero crear un modelo del sistema a construir. Ayudan también en la evaluación de la calidad del modelo. Mediante la comprobación de la consistencia y de la validez de cada modelo, proporcionan el conocimiento y la ayuda necesarios para eliminar errores antes de que se propaguen por el programa.

Herramientas de Programación

Son parte legítima de la tecnología CASE los utilitarios de software del sistema, los editores, los compiladores y los depuradores. Pero además se incluyen otras herramientas más nuevas y poderosas de programación tales como: la programación 0.0., los lenguajes de programación de cuarta generación, los sistemas avanzados de consultas a bases de datos, etc.

Herramientas de prueba e integración	Proporcionan muchos niveles de soporte diferentes para los pasos de prueba del software. Proporcionan durante las distintas etapas de prueba y sirven para diseñar casos de prueba (ej: analizadores de caminos) y ayudan a reducir el esfuerzo aplicado al proceso de prueba.
Herramientas de simulación y de creación de **prototipos**	Abarcan un amplio conjunto de herramientas de variada sofisticación, desde sencillos programas de dibujo hasta productos de simulación para el análisis de la temporización y el dimensionamiento de sistemas empotrados de tiempo real. Se centran en la creación de pantallas e informes que permitirán al usuario entender el ámbito de entrada y salida de un sistema de información o de una aplicación de ingeniería. A un nivel más sofisticado, permiten crear un modelo del sistema para aplicaciones empotradas de tiempo real.
Herramientas de mantenimiento	Ayudan al ingeniero a descomponer un programa existente, proporcionándole una visión general del mismo. Aunque, de todos modos, el proceso de ingeniería inversa y/o reingeniería requiere de una componente humana en cuanto a la intuición, el sentido del diseño y la inteligencia del ingeniero que hacen que difícilmente pueda ser en un futuro cercano un proceso totalmente automatizado.
Herramientas de estructura	Proporcionan una estructura a partir de la cual se puede crear un entorno integrado de soporte para el proyecto. Generalmente, son facilidades de gestión de bases de datos y gestión de configuraciones.

Reusabilidad

Es la creación y reutilización de bloques constructivos de software, que deben ser catalogados para un fácil referencia, estandarizados para un fácil aplicación y validados para una fácil integración.

Aunque existen actualmente bibliotecas de software reutilizable para aplicaciones comerciales, de sistemas y de tiempo real y para problemas científicos y de ingeniería, lo que no existe son técnicas sistemáticas para ampliar las bibliotecas, resulta difícil imponer estándares para las interfaces de software reutilizable, los aspectos de mantenibilidad y calidad están sin resolver, y a menudo, los equipos de desarrollo ignoran la existencia de bloques constructivos de software adecuados.

Estimación del proyecto de Software

Estimación del proyecto de Software

El software es el elemento más caro de la mayoría de los sistemas informáticos.

La estimación de costo y esfuerzo no es una ciencia exacta ya que hay demasiadas variables (humanas, técnicas, de entorno, políticas, etc.) que pueden afectar el costo final del software y el esfuerzo aplicado para desarrollarlo. Sin embargo la estimación del proyecto de software puede convertirse en una serie de pasos sistemáticos que proporcionen estimaciones con un grado de riesgo aceptable.

Para realizar estimaciones seguras de costos y esfuerzos, algunas de las opciones posibles son: **a)** Usar **técnicas de descomposición** sencillas, **b)** Desarrollar un **modelo empírico** o **c)** Adquirir una o varias herramientas automáticas de estimación.

Técnicas de Descomposición

Usan un enfoque **"divide and conquer"** para la estimación de un proyecto de software, mediante la descomposición del proyecto en sus funciones principales y en la tareas de ingeniería de software que les corresponden, donde la estimación de costo y del esfuerzo va haciéndose en forma escalonada idónea.

Independientemente de la variable de estimación que se use, LOO o PF, **el planificador del proyecto proporciona un rango de valores para cada función descompuesta**. A partir de los datos históricos, él estima valores **optimista, más probable y pesimista** de LDC o PF para cada función. Cuando lo que se proporciona es un rango de valores, implícitamente se proporciona una indicación del grado de incertidumbre.

Con este enfoque, natural, de resolución del problema de estimar el costo y el esfuerzo de un proyecto de software, se calcula el **valor esperado** de LDC o de PF.

planificador del proyecto proporciona un rango de valores para cada función descompuesta. A partir de los datos históricos, él estima valores **optimista, más probable y pesimista** de LDC o PF para cada función. Cuando lo que se proporciona es un rango de valores, implícitamente se proporciona una indicación del grado de incertidumbre.

NO VA

Con este enfoque, natural, de resolución del problema de estimar el costo y el esfuerzo de un proyecto de software, se calcula el **valor esperado** de LDC o de PF. El valor esperado para la variable de estimación, E, se obtiene como una media ponderada de las estimaciones de LDC o PF optimista (a), más probable (b) y pesimista (c).

$$E = \frac{a + 4b + c}{6}$$

que da una mayor credibilidad a la estimación más probable y sigue una distribución de probabilidad β.

Asumimos que la probabilidad de que el resultado real de LDC o de PF quede fuera del rango establecido por los valores optimista y pesimista es muy baja. Entonces, se puede calcular la **desviación de las estimaciones.** Sin embargo, hay que tener en cuenta que se trata de una desviación basada en datos inciertos, por lo que debe usarse con cuidado.

Una vez que se ha determinado el valor esperado para la variable de estimación, se aplican los datos de productividad para LDC o para PF de acuerdo a uno de los dos siguientes métodos.

1) Se puede **multiplicar el valor total de la variable de estimación para todas las subfunciones por la métrica de productividad media** correspondiente a la variable de estimación.

Por ejemplo, si en total se han estimado 310 PF y la productividad media de PF, de acuerdo a proyectos anteriores es de 5,5 PF/pm, entonces el esfuerzo total para el proyecto será:

$$\text{Esfuerzo} = \frac{310}{5,5} = 56 \text{ personas/mes}$$

2) **Se puede multiplicar el valor de la variable de estimación obtenida para cada subfunción por el *valor de productividad compensado,*** que se basa en el nivel de complejidad percibido para la subfunción. Para funciones de complejidad media se usa una métrica de productividad media. Para cada subfunción concreta se compensa la métrica de productividad media hacia arriba y hacia abajo dependiendo de que la complejidad sea mayor o menor que la media (bastante subjetivo).

Por ejemplo, si la productividad media fuera de 490 LDC/pm, se podría estimar la productividad para las subfunciones que sean considerablemente más complejas que la media en sólo 300 LDC/pm y para las funciones simples en 650 LDC/pm.

En el caso de **estimar el esfuerzo,** técnica más usada para calcular el costo del proyecto, se comienza también por la **delimitación de las funciones del software**, obtenidas del ámbito del proyecto. El planificador estima el esfuerzo (por ejemplo en personas/mes) de acometer cada tarea de ingeniería de software de cada función del software. Y arma con ellos una tabla del tipo:

	Tareas	Total
Funciones	Estimaciones en personas/mes	

Total
Costo ($)
Tarifa ($)

Al igual que las técnicas LDC y PF, la estimación del esfuerzo comienza con la delimitación de las funciones del software, obtenidas del ámbito del proyecto. Para cada función debe realizarse una serie de tareas de Ingeniería de software - análisis de requisitos, diseño, implementación y pruebas.

El **planificador estima el esfuerzo de acometer cada tarea de ingeniería de cada función del software**. Se aplican las tarifas laborales (costo/ unidad de esfuerzo) a cada tarea de ingeniería (En general, la tarifa laboral es diferente para cada tarea ya que el personal senior está involucrado en el análisis de requisitos y en las primeras etapas del diseño, mientras que el personal junior (menos costoso) se ocupa de etapas posteriores del diseño, en la implementación y pruebas iniciales).

Por último, se calculan los costos y el esfuerzo para cada función y cada tarea de ingeniería.

Modelos Empíricos de Estimación

Generalmente se usan como **complemento de las técnicas de descomposición**. Cada modelo se basa en la experiencia (datos históricos) **y toma** como base:

$$d = f(y_i)$$

Donde d es uno de los valores estimados (esfuerzo, costo, duración del proyecto) y los y_i son determinados parámetros independientes (Ej. : LDC o PF estimados).

En otras palabras, este modelo **usa fórmulas derivadas empíricamente para predecir los datos que se requieren en el paso de planificación del proyecto de software.** Los datos empíricos que soportan la mayoría de los modelos se obtienen de una muestra limitada de proyectos. Por esta razón, un mismo modelo es estimación no es adecuado para todas las clases de software ni para todos los entornos de desarrollo.

Los ***modelos de recursos*** consisten en una o varias ecuaciones obtenidas empíricamente que predicen el esfuerzo (en personas/mes), la duración del proyecto (en meses cronológicos) y otros datos relativos al proyecto.

Basili, 1980, describe **cuatro clases de modelos de recursos: *modelos univariable estáticos, modelos multivariable estáticos, modelos multivariable dinámicos y modelos teóricos.***

Un **modelo univariable estático** toma la forma:

$$\text{Recurso} = c_1 \times (\text{característica estimada})^{c_2}$$

Donde el recurso podría ser el esfuerzo, la duración del proyecto, la cantidad de personal o las líneas requeridas de documentación del software. Las constantes c_1 y c_2 se derivan de los datos recopilados de proyectos anteriores. La característica estimada puede ser la cantidad de líneas de código fuente, el esfuerzo (si ya está estimado) u otra característica del software.

Si hay suficientes datos históricos disponibles se pueden derivar modelos con la forma recién descripta. (Por ejemplo: la versión básica del modelo C000MO o modelo de coste constructivo>.

Los ***modelos multivariable estáticos*** también usan datos históricos para obtener relaciones empíricas. Un modelo típico de esta categoría toma la forma:

$$\text{Recurso} = c_{11}\, e, + c_{21}\, e_2 + \ldots\ldots$$

Donde e_i es la característica i-ésima del software y c_{i1}, c_{i2} son constantes obtenidas empíricamente para la características i-ésima.

Un ***modelo multivariable dinámico*** proyecta los requisitos de recursos como una función del tiempo. Si se obtiene empíricamente el modelo, los recursos se definen en una serie de pasos consecutivos en el tiempo que asignan cierto porcentaje de esfuerzo (o de otro recurso) a cada etapa del proceso de ingeniería de software. Además, cada paso debe ser subdividido en tareas.

El enfoque teórico de la modelización multivariable dinámica incluye una **"curva continua de utilización del recurso"** como hipótesis y, a partir de ella, obtiene ecuaciones que modelizan el comportamiento del recurso.

Cada uno de los modelos anteriores se centra en aspectos macroscópicos del desarrollo del proyecto de software. Este último modelo de recursos examina el software desde un punto de vista microscópico, es decir las características del código fuente (número de operadores y operandos, por ejemplo).

COCOMO

Barry Bohem, en su libro sobre economía de la ingeniería de software, presenta una **jerarquía de modelos de estimación para el software**, que recibe el nombre genérico de COCOMO (**Co**nstructive **Cost Mo**del), constituida por los siguientes:

Modelo 1 Es el modelo básico, univariable estático que calcula el esfuerzo (y el costo) del desarrollo de software en función del tamaño del programa, expresado en las LDC estimadas.

Modelo 2 Es el modelo intermedio que calcula el esfuerzo de desarrollo en función del tamaño del programa y de un conjunto de "constructores de costo" que incluyen la evaluación subjetiva del producto, del hardware, del personal y de los atributos del proyecto.

Modelo 3 Es el modelo avanzado que incorpora todas las características del modelo intermedio y lleva a cabo una evaluación del impacto de los constructores de costo en cada fase (análisis, diseño, etc.) del proceso de ingeniería

Los modelos COCOMO están definidos para tres tipos de proyectos de software: **(1) Proyectos del software relativamente pequeños y sencillos** en los que trabajan pequeños equipos, con buena experiencia en la aplicación, sobre un conjunto de requisitos poco rígidos (ej. : un programa de análisis termal desarrollado para un grupo calórico>. ***(Modo orgánico)*** **(2) Proyectos de software intermedios** (en tamaño y complejidad) en los que equipos con variados niveles de experiencia deben satisfacer requisitos poco o medio rígidos (ej. : un sistema de transacciones de requisitos fijos para hardware de terminal o software de gestión de bases de datos) ***(Modo semi-acoplado)*** y **(3) Proyectos de software que deben ser desarrollados en un conjunto de hardware,** software y restricciones operativas muy restringidos (ej. : software de control de navegación para un avión) ***(Modo empotrado).***

Las ecuaciones del Cocomo básico tienen la siguiente forma:

$$E = a_\beta [KLOC]^{b}_{\beta}$$
$$D = c_\beta (E)^{d}_{\beta}$$

donde:

E es el esfuerzo aplicado en personas – mes
D es el tiempo de desarrollo en meses cronológicos
KLDC es el número estimado de líneas de código para el proyecto

Los coeficientes y los exponentes se han tabulado de la siguiente manera:

Cocomo básico

Proyecto de software	a_β	b_β	c_β	d_β
Orgánico	2,4	1,05	2,5	0,38
Semi-acoplado	3,0	1,12	2,5	0,35
Empotrado	3,6	1,20	2,5	0,32

El modelo básico se amplia para considerar un conjunto de "atributos

conductores de costo" que pueden agruparse en cuatro categorías principales:

1. Atributos del Producto
a) Fiabilidad requerida del software
b) Tamaño de la base datos de la aplicación
c) Complejidad del producto

2. Atributos del hardware
a. Restricciones de rendimiento en tiempo de ejecución.
b. Restricciones de memoria.
c. Volatilidad del entorno de la máquina virtual.
d. Tiempo de espera requerido.

3. Atributos del personal
a. Capacidad de análisis
b. Capacidad del Ingeniero de software
c. Experiencia en aplicaciones
d. Experiencia con la máquina virtual
e. Experiencia con el lenguaje de programación

4. Atributos del proyecto
a. Utilización de herramientas de software
b. Aplicación de métodos de ingeniería de software
c. Planificación temporal del desarrollo.

Cada uno de los 15 atributos es valorado en una escala de 6 puntos que va desde "muy bajo" hasta "extra alto". De acuerdo con esta evaluación se determina un **multiplicador de esfuerzo a** partir de las tablas, y con el producto de todos los multiplicadores de esfuerzo, se obtiene un **factor de ajuste** del esfuerzo. Los valores típicos para el FAE van de 0,9 a 1 ,4.

La ecuación del modelo Cocomo intermedio toma la forma:

$$E = a_i(KLDC)^{b_i} \times FAE$$

donde:

E es el esfuerzo aplicado en personas – mes
LDC es el número estimado de líneas de código para el proyecto.

Los coeficientes y exponentes se muestra en la siguiente tabla:

Cocomo básico

Proyecto de software	a_i	b_i
Orgánico	3,2	1,05
Semi-acoplado	3,0	1,12
Empotrado	2,8	1,20

Cocomo es el modelo empírico más completo para la estimación del software publicado hasta la fecha aunque estima los costos de desarrollo en alrededor de un

20% de su valor real, y el tiempo en un 70%. Aunque esta no es la precisión deseada es una buena ayuda en el análisis económico de la Ingeniería de Software y en la toma de decisiones.

Herramientas automáticas de estimación

Estas herramientas implementan una o más técnicas de descomposición o modelos empíricos. Suelen ser, combinadas con una interfaz interactiva hombre – máquina, una opción atractiva para la estimación.

Conclusiones

El planificador de un proyecto tiene que estimar tres ítems antes de comenzar el proyecto:

a) Cuánto durará
b) Cuánto esfuerzo requerirá
c) Cuánta gente estará implicada

Además, **deberá predecir los recursos** (de hardware y software) que se van a requerir y el riesgo implicado.

La especificación del ámbito ayuda al planificador a desarrollar estimaciones mediante algunas de las siguientes técnicas:

i. Descomposición
ii. Modelización
iii. Herramientas automatizadas

Mediante la reconciliación y comparación de estimaciones obtenidas con las diferentes técnicas, el planificador puede obtener una estimación más precisa. Aunque la estimación nunca será exacta, la combinación de buenos datos históricos y de técnicas sistemáticas puede mejorar la eficacia de la estimación.

GESTIÓN DE RIESGOS

Ingeniería de Software
Ian Sommerville – 6ta Edición

Una tarea importante del gestor de proyectos es anticipar los riesgos que podrían afectar el desarrollo del proyecto o la calidad del software a desarrollar, y emprender acciones para evitar esos riesgos.

Los resultados del análisis de riesgos se deben documentar a lo largo del plan del proyecto junto con el análisis de consecuencias cuando el riesgo ocurra.

De una forma simple se puede definir:

Riesgo: **Probabilidad de que una circunstancia adversa ocurra. Los riesgos son una amenaza para el proyecto, para el software que se está desarrollando y para la organización.**

Estas categorías de riesgo se definen a continuación:

1. Los ***riesgos del proyecto*** afectan la calendarización o los recursos del proyecto.
2. Los ***riesgos del producto*** afectan la calidad o desempeño del software que se está desarrollando.
3. Los ***riesgos del negocio*** afectan a la organización que desarrolla el software.

Por supuesto, esta clasificación no es única. Si un programador experto abandona el proyecto, esto es un riesgo para éste (debido a que la entrega del sistema se puede retrasar), para el producto (debido a que el reemplazante puede no ser tan experto y cometer muchos errores) y para el negocio (debido a que esa experiencia puede no contribuir a negocios futuros).

La gestión de riesgos es importante particularmente para proyectos de software debido a las incertidumbres inherentes que enfrentan muchos proyectos, ya que se pueden: definir pobremente los requerimientos, las dificultades en la estimación de tiempos y recursos, la dependencia en las habilidades personales y los cambios en los requerimientos debido a cambios en las necesidades del cliente.

El administrador de proyectos debe anticiparse a los riesgo; comprender el impacto de éstos en el proyecto, en el producto y en el negocio; y considerara los pasos para evitarlos. En el caso de que ocurran se deben crear planes de contingencia para que sea posible aplicar acciones de recuperación.

Los tipos de riesgo que pueden afectar un proyecto dependen de éste y el entorno organizacional en que se esté desarrollando el mismo. Sin embargo, muchos riesgos son universales, tales como: Rotación de personal, Cambio de administración, No disponibilidad del hardware, cambio de requerimientos, retrasos en la especificación, subestimación de tamaño, bajo desempeño de la

herramienta CASE, cambio de tecnología, competencia del producto (un producto competitivo se pone en venta antes de que el sistema se complete).

El proceso de administración de riesgos y los resultados de cada paso son:

1. **Identificación de Riesgos → Listado de riesgos potenciales**

   Identificar los posibles riesgos para el proyecto, producto y negocios. Elaborar un listado de todos los riesgos identificados.

2. **Análisis de riesgos → Listado de priorización de riesgos**

   Valorar las probabilidades y consecuencias de estos riesgos

3. **Planificación de riesgos → Mitigación (anulación) de riesgos y planes de contingencia**

   Crear planes para abordar los riesgos, ya sea para evitarlos o minimizar sus efectos en el proyecto.

4. **Supervisión de riesgos → Valoración de riesgos**

   Valorar los riesgos en forma constante y revisar planes para mitigación de riesgos tan pronto como la información de riesgos esté disponible.

El proceso de gestión de riesgos es un proceso iterativo que se aplica a lo largo de todo el proyecto. Los riesgos se monitorizan, se re-analizan y se establecen nuevas prioridades. La prevención de riesgos y los planes de contingencia se deben modificar tan pronto como surja nueva información de riesgos.

Los resultados del proceso de gestión de riesgos se deben documentar en un plan de gestión de riesgos que incluya una discusión de los riesgos a los que se enfrenta el proyecto, un análisis de ellos y los planes requeridos para su gestión.

G.R.1. Identificación de riesgos

Es la primera etapa de la gestión de riesgos. Implica descubrir los posibles riesgos del proyecto. En principio se los debe valorar o priorizar en esta etapa, aunque en la práctica, en general, no se consideran los riesgos de baja probabilidad o con consecuencias menores.

Para la identificación se usan reuniones de grupo usando un enfoque de lluvia de ideas o simplemente basándose en la experiencia del administrador. Se pueden usar una lista de posibles tipos de riesgos:

1. ***Riesgos de tecnología***: Se derivan de las tecnologías de SW o de HW usadas en el sistema que se está desarrollando. Ej. La base de

datos que se usa en el sistema no puede procesar muchas transacciones por segundo como se esperaba; Los componentes de SW a reutilizarse contienen defectos que limitan su funcionalidad.

2. ***Riesgos de personas***: Asociados con las personas en el equipo de desarrollo. Ej. Es imposible reclutar personal con las habilidades requeridas para el proyecto; El personal clave no está disponible en momentos críticos; la capacitación solicitada del personal no está disponible.

3. ***Riesgos Organizacionales***: Derivan del entorno organizacional donde el software se está desarrollando. Ej. La organización se reestructura y una administración diferente se responsabiliza del proyecto; los problemas financieros de la organización fuerzan a reducciones en el presupuesto del proyecto.

4. ***Riesgos de Herramientas***: Se derivan de las herramientas CASE y otro SW de apoyo usado para desarrollar el sistema. Ej. El código generado por las herramientas CASE es ineficiente; las herramientas CASE no se pueden integrar.

5. ***Riesgos de requerimientos***: Se derivan de los cambios de los requerimientos del cliente y el proceso de administrar dicho cambio. Ej. Se proponen cambios en los requerimientos que requieren rehacer el diseño; Los clientes no comprenden el impacto de los cambios en los requerimientos.

6. ***Riesgos de estimación***: Se derivan de los estimados administrativos de las características del sistema y los recursos para construirlo. Ej. Se ha subestimado el tiempo requerido para desarrollar el software, o la tasa de reparación de defectos o el tamaño del software.

El resultado de este proceso debe ser una larga lista de riesgos que podrían ocurrir y afectar el producto, el proceso o el negocio.

G.R.2. Análisis de Riesgos

Durante este proceso se considera, por separado, cada riesgo identificado y se decide acerca de la probabilidad y la seriedad del mismo.

Para esta tarea se necesita la opinión y la experiencia del gestor de proyectos. Se hace una valoración en intervalos:

1. La probabilidad de que el riesgo ocurra se valora como: muy baja (menor que 10%), baja (entre 10 y 25%); moderado (entre 25 y 50%); alta (entre 50 y 75%) o muy alta (mayor que 75%).
2. Los efectos del riesgo, cuando ocurre, se valoran como: catastrófico, serio, tolerable o insignificante.

Los resultados de este análisis se colocan en una tabla que se orden de acuerdo a la seriedad del riesgo (para hacer la valoración se necesita información detallada del proyecto, proceso, el equipo de desarrollo y la organización).

Ya que la probabilidad y la valoración de los efectos del riesgo cambian conforme se disponga de mayor información acerca del riesgo y los planes de gestión de riesgos se implementen, la tabla se debe actualizar durante cada iteración del proceso de riesgos.

Una vez que los riesgos se hayan analizado u clasificado, se debe discernir cuáles son los más importantes a considerar durante el proyecto. Este discernimiento depende de una combinación de la probabilidad del riego en cuestión y los efectos del mismo. En general, se recomienda tomar en cuenta los riesgos catastróficos así como los riesgos serios que tienen más que una moderada probabilidad de ocurrir.

El número exacto de riesgos a supervisar debe depender del proyecto, pero debe ser manejable (5, 10 o 15). Pero debe ser un número manejable. Un número grande de riesgos requiere obtener mucha información.

G.R.3. Planificación de Riesgos

Este proceso considera cada uno de los riesgos claves identificados y las estrategias para administrarlo. No existe un proceso sencillo para establecer los planes de administración de riesgos, recae en el juicio y experiencia del administrador de proyectos.

Las estrategias caen en tres categorías:

a) ***Estrategias de anulación***: Seguir éstas significa reducir la probabilidad de que ese riesgo surja.
b) ***Estrategias de disminución***: Seguir éstas significa reducir el impacto del riesgo.
c) ***Planes de contingencia***: Seguir éstas significa que, si sucede lo peor, se está preparado para ello y se cuenta con una estrategia para abordarlo.

Algunos ejemplos de riesgos identificados y estrategias para abordarlos son:

Riego	**Estrategia**
Problemas financieros De la organización	Preparar un documento breve para el administrador que muestre que el proyecto hace contribuciones muy importantes a las metas del negocio. Ejemplo de c).
Problemas de Reclutamiento	Alertar al cliente de las dificultades potenciales y las posibilidades de retraso, investigar los componentes comprados.

Riego	Estrategia
Enfermedad del Personal	Reorganizar el equipo de tal forma que haya traslape en el trabajo y las personas comprendan el de los demás. Ejemplo de b).
Componentes Defectuosos	Reemplazar los componentes defectuosos con los comprados de fiabilidad conocida. Ejemplo de a).
Cambios en los Requerimientos	Rastrear la información para valorar el impacto de los requerimientos, maximizar la información oculta de ellos.
Reestructuración Organizacional	Preparar documento breve para el administrador principal que muestre que el proyecto hace contribuciones muy importantes a las metas del negocio.
Desempeño de la Base de Datos	Investigar la posibilidad de comprar una base de datos con alto desempeño.
Tiempo de desarrollo Subestimado	Investigar los componentes comprados y la utilización de un generador de programas.

G.R.4 Supervisión de riesgos

En este paso, normalmente, se valora cada uno de los riesgos identificados para decidir si éste es más o menos probable y cuándo los efectos del mismo han cambiado. Como no se puede observar en forma directa, se buscan otros factores para indicar la probabilidad del riesgo y sus efectos.

Algunos ejemplos se muestran en la siguiente tabla:

Tipo de Riesgo	Indicadores Potenciales
Tecnología	Entrega retrasada del Hardware o de la ayuda al software, mucho problemas tecnológicos reportados
Personas	Baja moral del personal, malas relaciones entre los miembros del equipo, disponibilidad de empleo.
Organizacional	Chismorreo organizacional, falta de acciones por el administrador principal.
Herramientas	Rechazo de los miembros del equipo para usar herramientas, quejas acerca de las herramientas CASE, peticiones de estaciones de trabajo más potentes.

Requerimientos	Peticiones de muchos cambios en los requerimientos Quejas del cliente.
Estimación	Fracaso en el cumplimiento de los tiempos acordados, y en la estimación de defectos reportados.

La supervisión de riesgos debe ser un proceso continuo, y en cada revisión del progreso de la administración, cada uno de los riesgos clave debe ser considerado por separado y discutido por la audiencia.

Entonces,

- Se deben identificar y valorar los riesgos mayores del proyecto para establecer su probabilidad y consecuencias para éste

- Para los riesgos más probables y potencialmente serios, se debe hacer planes para anularlos, administrarlos o tratarlos.

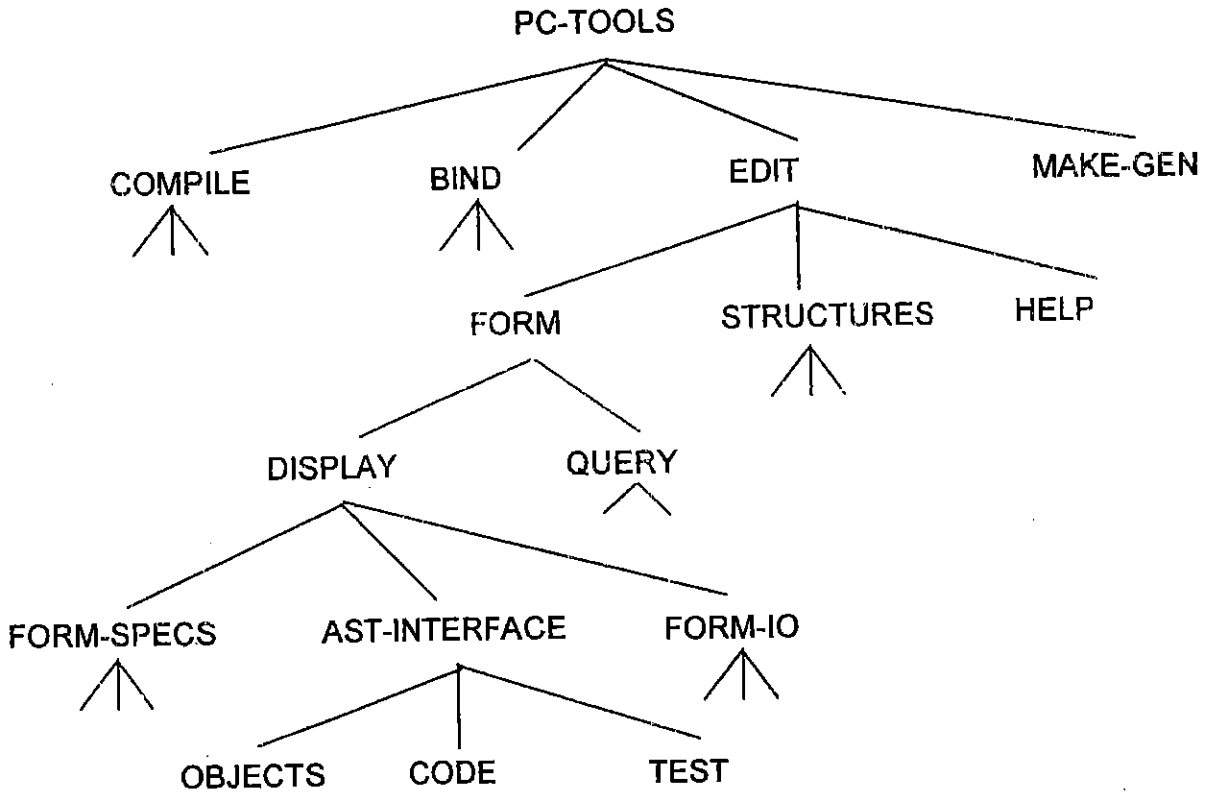
- Estos riesgos se deben discutir de manera explícita en cada reunión del progreso del proyecto.

Gestión de configuración

- Es esencial para asegurar que las versiones correctas de sub-sistemas y componentes se integren en el momento correcto.
- La gestión de la configuración implica
  - Identificación y almacenamiento de componentes.
  - Gestión del cambio.
  - Gestión de versiones.
  - Construcción del sistema.

Identificación y almacenamiento de componentes

- Los proyectos grandes típicamente producen miles de componentes que deben ser identificados.
- Algunos de estos componentes deben ser mantenidos durante todo el tiempo de vida del software.
- Se debe definir un esquema para nombrar componentes de manera tal que los componentes relacionados tengan nombres relacionados.
- Probablemente el enfoque más flexible es usar un esquema jerárquico con nombres multi-nivel.



Gestión del cambio

- El software está sujeto a continuos pedido de cambios durante el proceso de integración a medida que se descubren problemas que deben resolverse.
- La Gestión del cambio trata de la gestión de estos cambios y de asegurar que los cambios se implementen de la manera más costo-efectiva.
- La gestión del cambio debe incluir procesos para decidir qué problemas resolver, quién debe resolverlos y cuándo ellos deben ser reparados.

Gestión de versiones

- Inventar un esquema de identificación para las versiones del software.
- Planear cuándo se debe producir una nueva versión del software.
- Asegurar que los procedimientos y herramientas de gestión de versiones se apliquen adecuadamente.
- Planificar y distribuir nuevos releases de software.

Construcción del sistema

- Implica tomar las componentes de sistema y combinarlas en un único sistema ejecutable.
- Diferentes combinaciones de componentes permiten construir diferentes sistemas.
- Puede llevar varios días construir un sistema grande si todas sus componentes son compiladas y linkeadas al mismo tiempo.

Problemas con la construcción del sistema

- ¿Las instrucciones de construcción contienen todos los componentes requeridos?. La ausencia de uno normalmente debería ser detectado por el linker.
- ¿Se identificó la versión apropiada del componente?. Un sí construido con la versión incorrecta de un componente seguramente fallará post- entrega.
- ¿Están todos los archivos de datos disponibles?. La construcción no debería depender de archivos de datos "estándar" que varían de lugar en lugar.
- ¿Las referencias a archivos de datos dentro de los componentes son correctas?. Embeber nombres absolutos en el código siempre causa problemas ya que las versiones para nombrar varían de lugar en lugar.

- ¿El sistema se está construyendo para la plataforma correcta?. A veces, se debe construir para una versión específica de sistema operativo o configuración del hardware.
- ¿Se construyó para la versión correcta de compilador y otras herramientas de software especificadas?. Distintas versiones de compilador pueden generar código diferente y el componente compilado exhibirá comportamientos distintos.

Algunas recomendaciones sobre todo el capítulo

- Una buena gestión de proyecto es esencial para el éxito del mismo.
- La naturaleza intangible del software produce problemas de gestión.
- Los administradores tienen diversos roles, pero sus actividades más significativas son: planificación, estimación y scheduling.
- La planificación y la estimación son procesos iterativos que continúan a través de todo el proyecto.
- Una **milestone** del proyecto es un estado predecible donde algunos reportes formales del progreso se presentan a la administración.
- Los diagramas de actividades y de barras son representaciones gráficas del cronograma del proyecto.
- Las actividades de gestión de la configuración incluyen el nombrar componentes, gestión de versiones, control de cambio y construcción del sistema.

Complejidad
La complejidad inherente al software

Análisis y DiseñoOO
Grady Booch (1996)
2º. Edición – Madison – Eslesy/Díaz de Santos

Ya se sabe que algunos sistemas de software no son complejos. Son aplicaciones especificadas, construidas, mantenidas y utilizadas por la misma persona, que generalmente tienen un propósito muy limitado y un ciclo de vida muy corto.

Por otro lado, interesa mucho más los desafíos que plantea el desarrollo de lo que se llama **software de dimensión industrial**. Los sistemas de software de este tipo tienden a tener un ciclo de vida más largo, y a lo largo del tiempo muchos usuarios llegan a depender de su correcto funcionamiento.

La característica distintiva del software de dimensión industrial es que resulta sumamente difícil, si no imposible, para el desarrollador individual comprender todas las sutilezas de su diseño. Es decir **la complejidad de estos sistemas excede la capacidad intelectual humana**. Esta complejidad parece ser una característica esencial de todos los sistemas de software de gran tamaño, en el sentido de que se puede dominar pero nunca eliminar.

La complejidad inherente al software se deriva de cuatro elementos:

a) **La complejidad del dominio del problema**: Los problemas que se intentan resolver se componen de muchísimos requisitos que compiten entre sí, y más aún, que a veces se contradicen.

La complejidad externa surge de "desacoplamientos" que existen entre los usuarios de un sistema y sus desarrolladores; los primeros tienen grandes problemas para expresar clara y precisamente sus necesidades, tienen ideas vagas de lo que desean, los desarrolladores no son expertos en el dominio del problema que el usuario maneja, y ambos tienen visiones muy diferentes sobre la naturaleza del problema: Habitualmente los requisitos se explicitan en documentos de textos muy extensos, y difíciles de comprender.

Una complicación adicional es que estos requisitos cambian con frecuencia durante el desarrollo de un sistema de software.

Debido a que un gran sistema de software implica una inversión considerable, los sistemas no se desechan cada vez que cambian los requisitos. Mas bien, están previstos pera evolucionar en el tiempo.

Mantenimiento: Se corrigen errores.
Evolución: Se responde a requisitos que cambian.
Conservación Se siguen empleando medios extraordinarios para mantener en operación un elemento de software anticuado y decadente.

b) **La dificultad de gestionar el proceso de desarrollo:** La tarea fundamental del equipo de desarrollo es dar vida a una ilusión de simplicidad, para defender a los usuarios de esta vasta y arbitraria complejidad externa. Se hace lo posible para escribir menos código mediante mecanismos ingeniosos y potentes de desarrollo, así como de **la reutilización de marcos estructurales de desarrollo y código ya existente**. Esta gran cantidad de trabajo exige contar con un equipo de desarrolladores que idealmente debe ser pequeño para evitar problemas asociados con la complejidad de la comunicación y coordinación de grupos humanos muy numerosos. El reto al que siempre se enfrenta la dirección de un grupo de desarrolladores es mantener la unidad e integridad del diseño.

c) **La flexibilidad que se puede alcanzar a través del software:** El software ofrece la flexibilidad máxima por lo que un desarrollador puede expresar cualquier clase de abstracción. El desarrollador tiende a construir por sí mismo prácticamente todos los bloques fundamentales sobre los que se apoyan estas abstracciones de más alto nivel, en una industria en la que existen muy pocos estándares similares. Así, el desarrollo de software sigue siendo un negocio enormemente laborioso.

d) **Los problemas de caracterizar el comportamiento de sistemas discretos:** Los sistemas del mundo real son en general sistemas continuos, en los que no puede haber sorpresas ocultas ya que pequeños cambios en las entradas siempre producen cambios consecuentemente en las salidas. Por el contrario, todos los sistemas de software son sistemas discretos, descriptos en cada instancia por el estado en que se encuentran un gran número de variables, a las que afectan, a veces de manera insospechada, eventos externos. Por esta razón, los sistemas discretos se deben probar a fondo, aunque no exhaustivamente. Es decir, hay que concentrarse en un grado de confianza aceptable en lo que se refiere a su corrección.

Las consecuencias de la complejidad ilimitada

Un constructor pensaría raramente en construir un subsótano en un edificio que ya tiene construidas 100 plantas. Pero un usuario de un sistema de software

suele, con mucha frecuencia, solicitar cambios equivalentes.

Nuestro fracaso en dominar la complejidad del software lleva a proyectos retrasados, que exceden el presupuesto y que son deficientes respecto a los requisitos fijados. Esta situación, conocida como la crisis del software, se traduce en el desperdicio de recursos humanos así cómo en una considerable pérdida de oportunidad. Un porcentaje considerable del personal de desarrollo en cualquier organización debe muchas veces estar dedicado al mantenimiento o a la conservación de software geriátrico. Esta situación, dada la incidencia directa o indirecta que tiene el software, en la base económica de los países industrializados y considerando en qué medida puede el software ampliar la potencia del individuo, es inaceptable.

La estructura de sistemas complejos
Los cinco atributos de un sistema complejo

Un sistema complejo funciona correctamente sólo gracias a la actividad colaboradora de cada una de sus partes principales. Los sistemas complejos son jerárquicos, donde cada nivel de esta jerarquía representa un nivel de abstracción diferente, cada uno de los cuales se construye sobre otro y aún así es comprensible por sí mismos.

Basándose en un trabajo de Simon y Ando, Courtois sugiere lo siguiente:
Existen cinco atributos comunes a todos tos sistemas complejos:

1) Frecuentemente, la complejidad toma la forma de una jerarquía por lo cual un sistema complejo se compone de subsistemas relacionados que tienen a su vez sus propios subsistemas, y así sucesivamente, hasta que se alcanza algún nivel ínfimo de componentes elementales. (Sólo se puede comprender aquellos sistemas que tienen una estructura jerárquica).

2) La elección de qué componentes de un sistema son primitivos es relativamente arbitraria y queda, en gran medida, a decisión del observador.

3) Los enlaces internos de los componentes suelen ser más fuertes que los enlaces entre componentes. Este hecho tiene el efecto de separar la dinámica de alta frecuencia de los componentes - que involucra la interacción entre componentes - de la dinámica de baja frecuencia - que involucra la interacción entre componentes.

4) Los sistemas jerárquicos están compuestos usualmente de sólo unas pocas clases diferentes de subsistemas en varias combinaciones y disposiciones. Es decir, tienen patrones comunes.

5) Se encontrará que un sistema complejo que funciona ha evolucionado de un sistema simple que funcionaba. Un sistema complejo diseñado desde cero nunca funciona, y no puede parcharse para conseguir que lo haga. Siempre hay que partir de un sistema simple que funcione. (Los sistemas tienden a evolucionar en el tiempo, y esta evolución es más rápida si se parte de un sistema simple que funciona. Los objetos primitivos para un sistema complejo deben ser objetos usados, mejorados y probados con anterioridad).

La complejidad de los sistemas de software que hay que desarrollar se va incrementando, pero existen limitaciones básicas sobre la habilidad humana para enfrentarse a esa complejidad.

Imponiendo orden al caos
El rol de la descomposición

La técnica de dominar la complejidad se conoce desde tiempo remotos:*"Divide and Conquer"* (Dijkstra).

Cuando se diseña un sistema de software complejo, es esencial descomponerlo en partes cada vez más pequeñas, cada una de las cuales se puede refinar en forma independiente. Así, para entender un nivel dado de un sistema basta **comprender unas pocas partes**(no necesariamente, todas) a la vez. De esta manera. la descomposición inteligente ataca directamente la complejidad inherente al software forzando une división del espacio de estados del sistema.

Descomposición algorítmica: Casi todos los programadores han sido adiestrados para el diseño estructurado descendente, por lo que enfrentan esta descomposición como una simple descomposición algorítmica, en la que cada módulo representa un paso importante de algún proceso global.

La visión algorítmica enfatiza el orden los eventos

Descomposición orientada a objetos: En este tipo de descomposición se ve el mundo como un conjunto de agentes autónomos que colaboran para llevar a cabo un comportamiento de nivel superior. Las operaciones se asocian a objetos, y las llamadas a una operación provocan la creación de otros objetos. Cada objeto tiene su propio comportamiento único, y cada uno modela algún objeto del mundo real. De este modo, un objeto es una entidad tangible 'que muestra un comportamiento bien definido. Los objetos hacen cosas cuando se les envía un mensaje.

La visión orientada a objetos resalta los agentes que, o bien causan acciones o bien están sujetas a estas acciones.

El rol de la abstracción

Los humanos hemos desarrollado una técnica excepcionalmente poderosa para enfrentarnos a la complejidad: realizamos abstracciones. Si somos incapaces de dominar en su totalidad a un objeto completo, ignoramos sus detalles no esenciales y tratamos en cambio con un modelo generalizado e idealizado del objeto. Los objetos, en el mundo OO, como abstracciones de entidades del mundo real, representan un agrupamiento de información particularmente denso y cohesivo.

El rol de la jerarquía

Clasificando objetos en grupos de abstracciones relacionadas (clases) se llega a la distinción explicita de las propiedades comunes y distintas entre diferentes objetos, lo que luego ayudará a dominar su complejidad inherente.

La identificación de jerarquías en un sistema de software complejo no suele ser tarea fácil ya que requiere que se descubran patrones entre muchos objetos, cada uno de los cuales puede incluir algún comportamiento muy complicado. Pero una vez que se exponen estas jerarquías, se simplifica en gran medida. la estructura y la comprensión del sistema.

El diseño de sistemas complejos

En todas las ramas de la ingeniería, el diseño es:

La aproximación disciplinada que se usa para inventar la solución a algún problema, proveyendo así un camino, desde los requisitos hasta la implementación.

En la ingeniería del software, Mostow sugiere que el propósito del diseño es construir un sistema que:

a) Satisfaga determinada especificación funcional.
b) Se ajuste a las limitaciones impuestas por el medio de destino.
c) Respete requisitos implícitos o explícitos sobre rendimiento y uso de recursos.
d) Satisfaga criterios de diseño implícitos o explícitos sobre la forma del artefacto.
e) Satisfaga restricciones, sobre el propio proceso de diseño, tales como su longitud, costos, o herramientas disponibles para realizarlo.

El propósito del diseño es crear una estructura interna (o arquitectura) clara y relativamente simple. Un diseño es el producto final del proceso de diseño. El diseño conlleva un balance entre un conjunto de requisitos que compiten. Los productos del diseño son modelos que permiten razonar sobre las estructuras, hacer concesiones cuando los requisitos entran en conflicto y, en general, proporcionar un anteproyecto para la implementación.

La importancia de construir un modelo

Cada modelo dentro de un diseño describe un aspecto específico del sistema que se está considerando. Se busca construir modelos nuevos sobre modelos viejos en los que ya se tiene confianza. Los modelos nos ofrecen la oportunidad de fallar en condiciones controladas. Se evalúa cada modelo sobre situaciones previstas y sobre situaciones improbables y se lo modifica cuando falla para que se comporte del modo esperado o deseado.

Los elementos de los métodos de diseño de software

El diseño de sistemas complejos conlleva un proceso incremental e iterativo. Se han desarrollado muchos métodos de diseño diferentes, que a pesar de todo tienen elementos en común.

- Notación: El lenguaje par expresar cada modelo.
- Proceso: Las actividades que tienden a la construcción ordenada de los modelos del sistema.
- Herramientas: Elementos o artefactos que permiten construir el modelo.

Un buen método de diseño se basa en fundamentos teóricos sólidos, pero aún así ofrece grados de libertad para la innovación artística.

Los modelos del desarrollo orientado a objetos

Aunque no existe **"el mejor"** método de diseño se ha encontrado un gran valor en la construcción de modelos armados a partir de la descomposición orientada a objetos.

Aplicando diseño OO se crea software resistente al cambio y escrito con economía de expresión. Se logra un mayor nivel de confianza en la corrección del software a través de una división inteligente del espacio de estados y se reducen los riesgos inherentes al desarrollo de sistemas complejos de software.

El diseño OO ofrece un rico conjunto de modelos: Físico, Lógico, Estático y Dinámico que reflejan la importancia de plasmar las jerarquías de clases y de objetos del sistema que se diseña. También cubren el espectro de las decisiones de diseño relevantes que se consideran en el diseño de un sistema complejo, y así construir implementaciones que poseen los cinco atributos de los sistemas, complejos bien formados.

Resumen

- El software es complejo de manera innata; esta complejidad excede frecuentemente la capacidad intelectual humana.
- La tarea del equipo de desarrollo es la de crear una ilusión de sencillez.
- La complejidad, a menudo, toma forma de jerarquía: "es-un" y "parte-de".
- Generalmente, los sistemas .complejos evolucionan de formas intermedias estables.
- Mediante el uso de la descomposición; la abstracción y la jerarquía puede hacerse frente a las limitaciones en el conocimiento humano.
- Los sistemas complejos pueden analizarse centrándose ya sea en los objetos o en los procesos.
- El diseño OO es el método que conduce a una descomposición OO; el diseño OO define una notación y' un proceso para construir sistemas complejos, y ofrece un rico conjunto de modelos lógicos y físicos con los cuales se puede razonar sobre diferentes aspectos del sistemas que se está considerando.