TEMA 3: DISEÑO

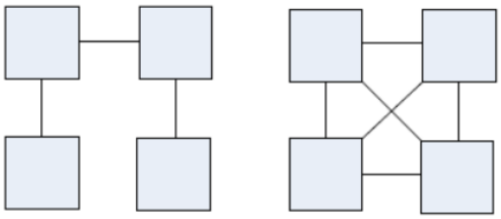
- El **diseño** puede definirse como
    - (1) el proceso para definir la arquitectura, los componentes, los interfaces, y otras características de un sistema o un componente,
    - (2) como el resultado de este proceso (“es decir, los planos o esquemas que muestran cómo debe ser construido el sistema”.)
- Un **componente**
    - es una parte funcional de un sistema que oculta su implementación proveyendo su realización a través de un conjunto de interfaces (“Se comunica con otros componentes a través de interfaces, sin que los demás tengan que saber cómo está construido”)
- Una **interfaz**
    - describe la frontera de comunicación entre dos entidades software/componentes, definiendo explícitamente el modo en que un componente interacciona con otros
    - (“ es la frontera de comunicación entre dos componentes de software. Define de forma clara cómo pueden interactuar entre sí: Qué datos pueden intercambiar, qué funciones o métodos están disponibles….
      Permite que un componente use otro sin necesidad de saber cómo está implementado.”)

Propiedades de diseño; Descomposición y modularización

- **Descomposición**: esta propiedad permite definir componentes de alto nivel en otros de bajo nivel., siguiendo la máxima de **«divide y vencerás»** hasta que cada módulo puede ser desarrollado individualmente.
- **Composición**: es el problema inverso a la descomposición. Un módulo de programación preserva la composición modular si facilita el diseño de elementos de programación que se pueden ensamblar entre sí para desarrollar aplicaciones.
- **Comprensión**: se preserva la comprensión modular si facilita el diseño de elementos de programación que se pueden interpretar fácilmente sin tener que conocer el resto de los módulos.
- **Continuidad**: un pequeño cambio en la especificación debe implicar un cambio igualmente pequeño en la implementación. Por ejemplo, es importante que los cambios en los requisitos repercutan en un número limitado y localizado de módulos.
- **Protección**: Según esta propiedad, los efectos de las anomalías de ejecución han de quedar confinados al módulo donde se produjo el error, o a un número limitado de módulos con los que éste interacciona directamente

Medición de la modularidad. Acoplamiento

- El acoplamiento mide el **grado de interconexión existente entre los módulos** en los que se ha dividido el diseño de la arquitectura de un sistema software.
- El objetivo es conseguir un **acoplamiento bajo** –débil–, pues genera sistemas fáciles de entender, mantener y modificar.
- Con un acoplamiento débil entre componentes los cambios en la interfaz de un componente afectarían un número reducido de cambios en otros (continuidad)



Medición de la modularidad. Cohesión

- Un subsistema o módulo tiene un alto grado de cohesión si **todos sus elementos mantienen una funcionalidad común.**
    - Por ejemplo, una clase dedicada al manejo de fechas, tiene sólo operaciones relacionadas con las fechas y no otras funcionalidades.
- El objetivo es diseñar módulos robustos y **altamente cohesionados** cuyos elementos estén fuertemente relacionados entre sí buscando la cohesión funcional, en la que un módulo realiza operaciones bien definidas y suscritas a una funcionalidad requerida, facilitando la comprensión y potenciando su reutilización.
- **Cuando los módulos agrupan elementos por otros motivos que no sean de estricta funcionalidad, la cohesión no será óptima.**
    - Ejemplos de mala cohesión serían la **cohesión secuencial,** en la que la funcionalidad se agrupa por la secuencia de ejecución, la **cohesión por comunicación**, donde los elementos se agrupan por compartir los mismos datos, o la peor de todas, la **cohesión por coincidencia**, donde se incluyen funcionalidades sin ningún orden (cajón desastre)

Arquitectura de sistemas

- La **arquitectura** de un sistema software es la organización fundamental de dicho sistema plasmada en sus componentes, las relaciones entre éstos y con el entorno, y los principios que guían su diseño e implementación.
- **Estilos arquitectónicos**. Cada uno de los diferentes modelos de representación conceptual en que se pueden organizar los componentes dentro de una arquitectura de software.
    - (“modelo o patrón que define cómo se organiza un sistema, es decir, una forma estándar de estructurar componentes. Ej: cliente-servidor, arquitectura en capas…”)

Estilos arquitectónicos

1.Filtro-tubería

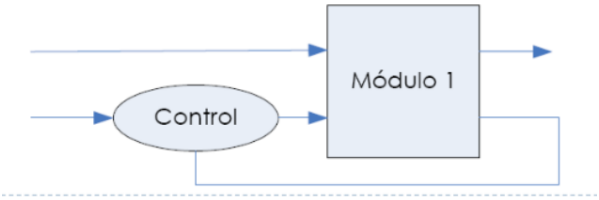
- Modelo basado en los flujos de datos, donde un componente transforma una entrada en una salida que a su vez es la entrada para otro componente.

2.Arquitecturas basadas en capas.

- Los componentes están organizados jerárquicamente por capas, donde cada capa provee servicios a la capa inmediatamente superior recibe servicios de la capa inmediatamente inferior.

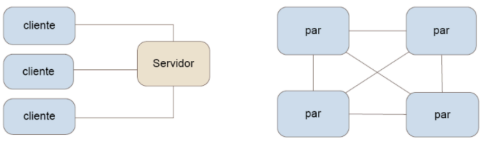
3.Control de procesos.

- Conceptualmente basados en los sistemas de control de los procesos industriales, este estilo asocia componentes de control –un bucle de control (feedback loop) en el modelo más simple– bien a la entrada o bien a la salida de un determinado proceso para supervisar su comportamiento y asegurar así que el comportamiento de dicho componente es el esperado.



4. Sist. distribuidos

- La funcionalidad global del sistema se divide en diferentes procesos generalmente distribuidos en diferentes máquinas.
- Se pueden diferenciar arquitecturas distribuidas cliente-servidor , y peer-to-Peer (P2P) por ejemplo con sistemas de distribución de ficheros TV IP o voz sobre IP (VoIP)



(otros:)

5. Sistemas basados en repositorios o de datos compartidos.

- Existe una estructura central de datos, independiente de los componentes, a la cual acceden los distintos componentes, los cuales operan en función de los datos del mismo. Es un modelo que permite la integración de distintos agentes, siendo el estado de los datos del repositorio el criterio de selección para el siguiente proceso a ejecutar.

6. Arquitectura basada en eventos (invocación implícita).

- “Es una arquitectura donde los componentes reaccionan a eventos, en lugar de llamarse directamente entre ellos. Los componentes no se invocan explícitamente. En su lugar, "escuchan" o "se suscriben" a ciertos eventos y actúan cuando estos ocurren”

La Orientación a Objetos (OO)

- El paradigma de la orientación a objetos se remonta a 1967, donde los noruegos Ole-Johan Dahl (1931-2002) y Kristen Nygaard (1926- 2002) desarrollaron los conceptos básicos de la programación orientada a objetos, en un lenguaje llamado SIMULA 67.
- Ambos recibieron conjuntamente el premio ACM Turing y la medalla IEEE John von Neumann por sus ideas seminales en la orientación a objetos.
- En la actualidad existe el premio Dahl-Nygaard de la asociación AITO (Association Internationale pour les Technologies Objets)
- Lenguajes influenciados por SIMULA 67: Smalltalk, Objective C, C++, Eiffel, CLOS, etc

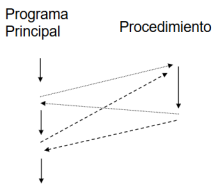
La evolución de los lenguajes ha sido:
Programación no estructurada-> Progra procedural-> Progra modular-> POO

1.Programación no estructurada

Datos son globales y pueden ser accedidos y modificados en cualquier punto de la ejecución de un programa

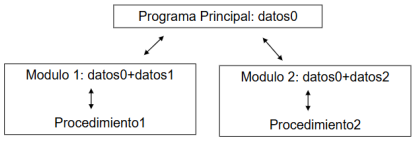
2.Programación procedural

Programa estructurado con subrutinas, y las subrutinas pueden ser llamadas desde cualquier punto del programa



3.Programación modular

- Procedimientos (subrutinas/funciones) con funcionalidad común están agrupados
- Cada módulo tiene su propio estado interno donde la interacción ocurre entre el programa principal y módulos a través de las llamadas a procedimientos

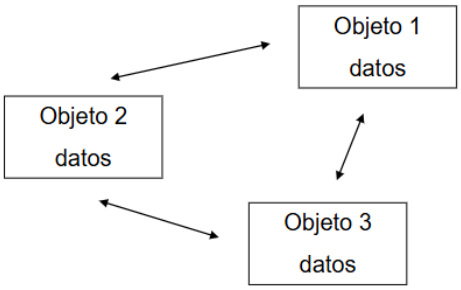


Problemas de la programación modular

- Problemas al modelar aplicaciones con datos no estructurados (“como objetos complejos, flujos de eventos …”) , ya que presuponen la modificación de entradas bien definidas en salidas igualmente estructuradas.
- Difícil reutilización de código, al manipular datos muy dependientes del dominio
- Difícil mantenimiento, por el mismo motivo que el anterior
- Reducción de criterios como eficiencia, flexibilidad, robustez, etc. con el mantenimiento (“Cada vez que haces cambios, mejoras, correcciones o ampliaciones al sistema (es decir, mantenimiento evolutivo o correctivo)”)

4. Programación OO

- Grafo de objetos cada uno con su propio estado
  - Un objeto es cualquier entidad que pueda ser modelada.
  - (Se definirá en detalle más adelante.)
- Los objetos interactúan entre ellos mediante envío de mensajes



¿Por qué la OO?

- Los conceptos de objeto son básicos para la mayoría de productos de software actuales:
  - Entornos gráficos (GUI) y de interacción con el ordenador, componentes, frameworks, etc, (son objetos)
- Las metodologías OO utilizan técnicas tradicionales de análisis del software pero:
  - La OO integra mucho mejor el análisis y el diseño. Proximidad de los conceptos de modelación respecto de las entidades (“Los objetos que modelas en el análisis (ej. “Usuario”, “Pedido”) se convierten directamente en clases del código.”)
  - Herramientas de generación de código OO automático desde entornos visuales:
    - **Herramientas CASE** (Computer Aided Software Engineering)
    - MDA Model Driven Architecture

Propiedades OO.

1. Abstracción

- La abstracción reduce la complejidad del dominio abstrayendo hasta el nivel adecuado. “La abstracción permite modelar entidades del mundo real extrayendo sus características comunes para definir clases generales que representen un conjunto de objetos similares”
- En la orientación a objetos la abstracción se representa mediante el concepto de clase, que representa un conjunto de objetos concretos, llamados instancias, que tienen propiedades y operaciones comunes.

2. Herencia

- La herencia permite definir una clase a partir de otras –una o más– clases existentes, de modo tal que la nueva clase hereda las características de la(s) superclase(s), a las que se añadirán ciertas características propias.

3. Encapsulamiento/encapsulación

- Consiste en **agrupar todos los datos y operaciones relacionadas en una misma clase**. Esta propiedad facilita que aparezcan otras características de la programación orientada a objetos como la **reutilización** y la **ocultación de información**.
- De esta manera, y debido a la ocultación de la información, los usuarios de esta clase **disponen de unos métodos que permiten consultar** y modificar el comportamiento de esta clase, pero **no tienen acceso directo a los datos.**
- **“Es el principio de ocultar los detalles internos de un objeto y proteger sus datos, permitiendo el acceso sólo a través de métodos públicos definidos.”**

4. Polimorfismo

- Propiedad por la que una operación se comporta de forma diferente en la **misma** o **diferentes clases por medio de la herencia**, por tanto existe la capacidad de que un mensaje sea interpretado de maneras distintas según el objeto que lo recibe.
- El polimorfismo en los lenguajes OO se observa en:
    - Atributos con el mismo nombre (en distintas clases)
    - Métodos con el mismo nombre (en la misma o en distintas clases)
    - Capacidad de ciertos elementos para recibir distintos objetos mediante una sola definición.

```
var shapes = new List<Shape>{
   new Rectangle(),
   new Triangle(),
   new Circle()};
foreach (var shape in shapes){
   shape.Draw();}
```

Unified Modeling Language - UML

- UML (Unified Modeling Language) es un **lenguaje de propósito general** que permite **visualizar, especificar, construir y documentar sistemas software**.
- Especialmente indicado para el modelado orientado a objetos.
- Estándar de Object Management Group

- UML combina notaciones provenientes de:
    - Modelado Orientado a Objetos, Modelado de Datos, Modelado de Componentes, Modelado de Flujos de Trabajo (Workflows)

Métricas.

1.Complejidad Estructural

- **Fan-in**, o grado de dependencia de un módulo, es el **número de módulos que llaman a dicho módulo.**
    - Un valor alto de fan-in indica que el módulo está **fuertemente acoplado (alto acoplamiento)**, por lo que los cambios en el mismo afectarán al resto del sistema (sensible a cambios).
- **Fan-out,** o grado de responsabilidad de coordinación de un módulo, es el **número de módulos que son llamados por dicho módulo.**
    - Valores altos de fan-out pueden indicar **módulos complejos** + alto acoplamiento debido a la complejidad de la lógica de control necesaria para coordinar las llamadas al módulo, "y mayor riesgo de errores si esos módulos de los que depende cambian".

2. Henry y Kafura

- Henry y Kafura (1981) enunciaron una conocida métrica de la complejidad estructural

$$HK_m = C_m\,(fan\text{-}in_m \cdot fan\text{-}out_m)2$$

donde Cm es la complejidad del módulo generalmente medida en líneas de código (LoC).

("Si el valor es muy alto, el módulo: Es grande (muchas líneas), Está muy acoplado (tiene muchas dependencias), Es difícil de mantener o modificar")

3.Otras

Dentro de la orientación a objetos, las métricas más ampliamente difundidas son:

- **MOOD** (Metrics for Object Oriented Design) definidas por Brito-Abreu y Melo (1996).
    - Proporción de métodos/atributos ocultos (privados…) respecto al total (mide encapsulación y protección de datos internos)
    - Proporción de métodos/atributos heredados (mide reutilización)
    - Proporción de polimorfismo (mide uso de polimorfismo)
- **Chidamber y Kemerer** (1994).
    - Acoplamiento entre objetos (CBO); "Número de clases a las que una clase está acoplada → bajo es mejor."
    - Respuesta para una clase (RFC): "Número total de métodos que pueden ser ejecutados por una clase → mide complejidad funcional."
    - Profundidad del árbol de herencia (DIT): "Longitud del camino desde la clase hasta la raíz de la jerarquía → puede indicar reutilización o complejidad."
    - Número de descendientes (NOC): "Cuántas clases heredan directamente → puede indicar potencial de reutilización o sobrecarga."

- Falta de cohesión en los métodos (LCOM): “Mide cohesión interna: cuántos métodos acceden a los mismos atributos. Si es alta, la clase está mal diseñada”

Diseño con patrones

- Un patrón de diseño es una forma de reutilizar el conocimiento abstracto sobre un problema y su solución.
- Un patrón es una descripción del problema y la esencia de su solución.
- Debe ser lo suficientemente abstracto como para ser reutilizado en diferentes configuraciones.
- “Un patrón de diseño es una solución reutilizable y general para resolver un problema común que aparece frecuentemente durante el diseño de software.”
- Las descripciones de patrones usualmente utilizan características orientadas a objetos como la herencia y el polimorfismo.

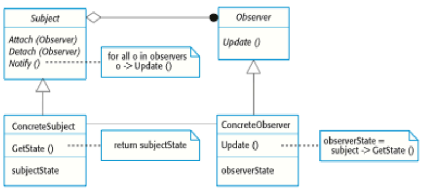
Elementos de los patrones

- Nombre: Un identificador de patrón significativo.
- Descripción del problema.
- Descripción de la solución: No es un diseño concreto, sino una plantilla para una solución de diseño que puede ser instanciada de diferentes maneras.
- Consecuencias: Los resultados y las compensaciones de la aplicación del patrón. (“Ventajas (desacoplamiento, reutilización, claridad), desventajas o compromisos: puede hacer el diseño más complejo, puede impactar el rendimiento, etc. Impacto en la extensibilidad, mantenibilidad o rendimiento. Relaciones con otros patrones (patrones relacionados o complementarios)....”)

Ej:

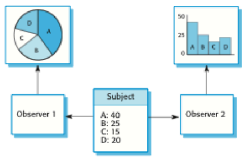
Nombre del patron	Observer
Descripción	Separa la visualización del estado de un objeto del objeto en sí y permite que se proporcionen pantallas alternativas. Cuando el estado del objeto cambia, todas las pantallas se notifican y actualizan automáticamente para reflejar el cambio.
Descripción del problema	En muchas situaciones, debe proporcionar varias pantallas de información de estado, como una pantalla gráfica y una pantalla tabular. No todos estos pueden ser conocidos cuando se especifica la información. Todas las presentaciones alternativas deben admitir la interacción y, cuando se cambia el estado, todas las pantallas deben actualizarse. Este patrón se puede usar en todas las situaciones donde se requiere más de un formato de visualización para información de estado y donde no es necesario que el objeto que mantiene la información de estado conozca los formatos de visualización específicos utilizados.

Modelo UML del patrón “Observer”



Pattern name	Observer
Descripción de la solución	Esto involucra dos objetos abstractos, Sujeto y Observador, y dos objetos concretos, ConcreteSubject y ConcreteObject, que heredan los atributos de los objetos abstractos relacionados. Los objetos abstractos incluyen operaciones generales que son aplicables en todas las situaciones. El estado que se muestra se mantiene en ConcreteSubject, que hereda las operaciones de Sujeto, lo que le permite agregar y eliminar observadores (cada observador corresponde a una pantalla) y emitir una notificación cuando el estado ha cambiado. ConcreteObserver mantiene una copia del estado de ConcreteSubject e implementa la interfaz Update () de Observer que permite que estas copias se mantengan en el paso. ConcreteObserver muestra automáticamente el estado y refleja los cambios cada vez que se actualiza el estado.
Consecuencias	El sujeto solo conoce al observador abstracto y no conoce detalles de la clase concreta. Por lo tanto, hay un mínimo acoplamiento entre estos objetos. Debido a esta falta de conocimiento, las optimizaciones que mejoran el rendimiento de la pantalla no son prácticas. Los cambios en el tema pueden hacer que se genere un conjunto de actualizaciones vinculadas a los observadores, algunas de las cuales pueden no ser necesarias.

Múltiples pantallas utilizando el patrón Observer



Observador: Cuando un objeto cambia su estado, todos los objetos que dependen de él son notificados y actualizados automáticamente. Un objeto (llamado Sujeto o Subject) necesita informar a varios otros objetos (Observadores) sobre un cambio. Util si quieres desacoplar al emisor de los receptores

Problemas de diseño típicos

- Para usar patrones en su diseño, debemos **reconocer que cualquier problema de diseño que esté enfrentando puede tener un patrón asociado que se pueda aplicar:**
  - Indicar a varios objetos que el estado de algún otro objeto ha cambiado **(patrón de observador)**.
  - Ordenar las interfaces para una serie de objetos relacionados que a menudo se han desarrollado de forma incremental **(patrón de Façade).** (“Tu sistema tiene muchas clases y módulos que exponen operaciones, y quieres una forma más simple y unificada de interactuar con ellos.”)
  - Proporcionar una forma estándar de acceder a los elementos de una colección, independientemente de cómo se implementa esa colección **(patrón de iterador).** (“Tienes una colección (lista, árbol, conjunto, etc.) y quieres acceder a sus elementos sin preocuparte por cómo están implementados.”)
  - Permitir la posibilidad de ampliar la funcionalidad de una clase existente en tiempo de ejecución **(patrón Decorator).**(“Necesitas añadir comportamiento nuevo a una clase, pero no puedes o no quieres modificarla directamente.”)

Aspectos de la implementación

El enfoque aquí no está en la programación, aunque esto es obviamente importante, sino en otros problemas de implementación que a menudo no se tratan en los textos de programación:

- **Reutilización**: La mayoría del software moderno se construye reutilizando componentes o sistemas existentes. Cuando estemos desarrollando software, debemos hacer el **mayor uso posible del código existente.**
- **Administración de la configuración**: **Durante el proceso de desarrollo**, debemos realizar un **seguimiento de las diferentes versiones de cada componente** de software en **un sistema de administración de la configuración.** (git..)

- **Desarrollo de destino de host:** El software creado generalmente no se ejecuta en la misma máquina que el entorno de desarrollo de software. Más bien, lo desarrollamos en un ordenador (el sistema host) y lo ejecutamos en un ordenador distinto (el sistema de destino).

Reuso/reutilización de software

- En la mayoría de las disciplinas de ingeniería, los sistemas se diseñan al juntar componentes existentes que se han utilizado en otros sistemas.
- La ingeniería de software se ha centrado más en el desarrollo original, pero ahora se reconoce que para lograr un **mejor software (al estar ya probado es menos propenso a fallos), más rápido y a menor costo,** necesitamos un proceso de diseño basado en la reutilización sistemática del software.

Ha habido un cambio importante al desarrollo basado en la reutilización en los últimos 10 años:

- Desde la década de 1960 hasta la década de 1990, la mayoría del nuevo software se desarrolló desde cero, al escribir todo el código en un lenguaje de programación de alto nivel.
    - La única reutilización o software significativo fue la **reutilización de funciones y objetos en bibliotecas de lenguaje** de programación.
- Los **altos costos** y la **presión de reducir tiempos** de desarrollo hicieron este enfoque cada vez más inviable, especialmente para los sistemas comerciales y basados en Internet.
    - Surgió un **enfoque de desarrollo basado en la reutilización de software existente** que ahora se utiliza generalmente para software empresarial y científico

Cosas a reutilizar:

- Reutilización de una aplicación: el conjunto de software de una aplicación puede reutilizarse ya sea incorporándolo sin cambios en otros sistemas o desarrollando familias de aplicaciones.
- Reutilización de componentes: los componentes de una aplicación de subsistemas de objetos individuales pueden ser reutilizados.
- Reutilización de objetos y funciones: los componentes de software que implementan un único objeto o función bien definidos pueden ser reutilizados

Reuso de productos comerciales (COTS)

- Un producto comercial (COTS, commercial-off-the-shelf) es un sistema de software que puede adaptarse a diferentes clientes sin cambiar el código fuente del sistema (del COTS).
- Los sistemas COTS tienen características genéricas y por lo tanto se pueden usar / reutilizar en diferentes entornos.
- Los productos COTS se adaptan mediante el uso de mecanismos de configuración incorporados que permiten que la funcionalidad del sistema se adapte a las necesidades específicas de los clientes. (“se adaptan mediante mecanismos de configuración

incorporados por el proveedor del software: Plantillas personalizables, Formularios modificables, Parámetros de entorno o de comportamiento, Módulos activables o desactivables, Interfaces para integrarse con otros sistemas (APIs)...")

  - Por ejemplo, en un sistema de registro de pacientes del hospital, se pueden definir formularios de entrada e informes de salida separados para diferentes tipos de pacientes

Niveles de reuso

- El nivel de abstracción.
  - En este nivel, no reutiliza el software directamente, sino que utiliza el conocimiento de las abstracciones exitosas en el diseño de su software.("No reutilizas código, sino ideas y estructuras de diseño exitosas")
- El nivel de objeto
  - En este nivel, reutiliza directamente los objetos/clases/funciones de una biblioteca en lugar de escribir de nuevo el código.
- El nivel de componente
  - Los componentes son colecciones de objetos y clases de objetos que se reutilizan en los sistemas de aplicaciones. ("reutilizar módulos de autenticación, componente carrito de la compra con todo lo que tiene…")
- El nivel del sistema
  - En este nivel, reutiliza sistemas de aplicación completos ("cots")

Costes del reuso

- Los **costes del tiempo** empleado en buscar software para reutilizarlo y evaluar si satisface sus necesidades o no.
- En su caso, los **costes de compra** del software reutilizable. Para grandes sistemas estos costos pueden ser muy altos.
- Los **costes de adaptar y configurar** los componentes o sistemas de software reutilizables para reflejar los requisitos del sistema que se está desarrollando.
- Los **costos de integrar elementos** de software reutilizables entre sí (si está utilizando software de diferentes fuentes) y con el nuevo código que se ha desarrollado.

Plataformas de desarrollo y de ejecución

- La mayoría del software se desarrolla en un ordenador (el host), pero se ejecuta en una máquina separada (el objetivo). Más generalmente, podemos hablar de una plataforma de desarrollo y una plataforma de ejecución.
- **Una plataforma es más que un simple hardware.** Incluye el sistema operativo instalado y otro software de soporte, como un sistema de gestión de bases de datos o, para plataformas de desarrollo, un entorno de desarrollo interactivo.
- La plataforma de desarrollo generalmente tiene un **software instalado diferente** al de la plataforma de ejecución; Estas plataformas **pueden tener diferentes arquitecturas**

Herramientas de la plataforma de desarrollo.

- Un **compilador integrado y un sistema de edición dirigido por sintaxis** que le permite crear, editar y compilar código. (VSC, netbeans…)

- **Un sistema de depuración de lenguaje.** (GDB…)
- **Herramientas de edición gráfica,** como herramientas para editar modelos UML. (enterprise architect)
- **Herramientas de prueba**, que puede ejecutar automáticamente un conjunto de pruebas en una nueva versión de un programa. (pytest, junit)
- **Herramientas de soporte de proyectos** que le ayudan a organizar el código para diferentes proyectos de desarrollo. (git+github, maven…)

Entornos de desarrollo integrado (IDE)

- Las herramientas de desarrollo de software a menudo se agrupan para crear un entorno de desarrollo integrado (IDE, integrated development environment).
- Un IDE es un **conjunto de herramientas de software que admite diferentes aspectos del desarrollo de software, dentro de un marco común y una interfaz de usuario.**
- Los IDE se crean para admitir el desarrollo en un lenguaje de programación específico, como Java. El IDE de un lenguaje particular se puede desarrollar especialmente, o puede ser una instanciación de un IDE de propósito general, con herramientas específicas de soporte de lenguaje (“instalando módulos específicos que permitan que sirva de IDE) (pycharm, intelliJ, netbeans..)

Factores de despliegue de componente y de sistema

- Si un componente está **diseñado para una arquitectura** de hardware específica, o se basa en algún otro sistema de software, obviamente debe implementarse en una plataforma que brinde el soporte de hardware y software requerido.
- Los **sistemas de alta disponibilidad** pueden requerir que los componentes se implementen en más de una plataforma. Esto significa que, en caso de falla de la plataforma, una implementación alternativa del componente esté disponible.
- Si hay un **alto nivel de tráfico** de comunicaciones entre los componentes, generalmente tiene sentido implementarlos en la misma plataforma o en plataformas físicamente cercanas entre sí. Esto reduce la demora entre el momento en que un mensaje envía un mensaje y otro lo recibe

Desarrollo de código abierto

El desarrollo de código abierto es un **enfoque del desarrollo de software** en el que **se publica el código fuente de un sistema** de software y **se invita a voluntarios a participar en el proceso de desarrollo.**

- Sus raíces se encuentran en la Free Software Foundation (www.fsf.org), que defiende que el código fuente no debe ser propietario, sino que siempre debe estar disponible para que los usuarios lo examinen y modifiquen según lo deseen.
- El software de fuente abierta extendió esta idea utilizando Internet para reclutar una población mucho mayor de desarrolladores voluntarios.
    - Muchos de ellos también son usuarios del código.

Sistemas de código abierto

- El producto de código abierto más conocido es, por supuesto, el sistema operativo Linux, que se utiliza ampliamente como sistema de servidor y, cada vez más, como entorno de escritorio.
- Otros productos importantes de código abierto son Java, el servidor web Apache y el sistema de administración de bases de datos mySQL.

¿Debería el producto utilizar componentes de código abierto? (chatgpt:)

Ventajas:

- Reducción de costes: no se paga por licencias.
- Aceleración del desarrollo: ya existen librerías, frameworks y módulos reutilizables.
- Flexibilidad: puedes modificar el código según tus necesidades.
- Comunidad activa: muchas veces hay foros, documentación y actualizaciones frecuentes.

Desventajas:

- Problemas de compatibilidad o integración con otros sistemas.
- Soporte limitado o informal, dependes de foros o voluntarios.
- Riesgos de seguridad si el componente no está bien mantenido o si encuentran vulnerabilidades
- Licencias complejas (ej. GPL puede obligarte a liberar tu propio código si lo usas mal).

¿Debería utilizarse un enfoque de código abierto para el desarrollo del software? (cgpt:)

Ventajas:

- Colaboración global: cualquier desarrollador puede contribuir.
- Transparencia: el código es abierto, auditable. Cualquiera puede ver el código fuente y revisar si tiene errores, fallos de seguridad o "puertas traseras"
- Rapidez en la innovación: mejora continua a través del feedback comunitario.
- Fomento de estándares abiertos y reutilización.

Desventajas:

- Pérdida de control total: cualquiera puede copiar o modificar el código.
- Difícil monetización directa si no tienes un modelo de negocio claro.
- Gestión de contribuciones: requiere moderar aportes y mantener calidad.
- Compromisos legales si hay contribuciones de terceros mal licenciadas. Si alguien te manda un fragmento de código que usa una licencia incompatible o tiene derechos de autor, puedes meterte en problemas legales sin saberlo.

Negocios con código abierto

- Cada vez más empresas de productos utilizan un enfoque de código abierto para el desarrollo.
- Su modelo de negocio no depende de la venta de un producto de software sino de la venta de soporte para ese producto (“capacitación para usarlo, mantenimiento, instalación, garantías de funcionamiento…”).
- Ellos creen que involucrar a la comunidad de código abierto permitirá que el software se desarrolle de manera más económica, más rápida y creará una comunidad de usuarios para el software.

Licencia de código abierto

- Un principio fundamental del desarrollo de código abierto es que el código fuente debería estar disponible de forma gratuita, **esto no significa que cualquiera pueda hacer lo que quiera con ese código.**
- Legalmente, el desarrollador del código (ya sea una empresa o un individuo) aún es propietario del código. Pueden imponer restricciones sobre cómo se usa al incluir condiciones legalmente vinculantes en una licencia de software de código abierto.
- Algunos desarrolladores de código abierto creen que si un componente de código abierto se usa para desarrollar un nuevo sistema, ese sistema también debería ser de código abierto.
- Otros están dispuestos a permitir que su código sea utilizado sin esta restricción. Los sistemas desarrollados pueden ser propietarios y vendidos como sistemas de código cerrado.

Modelos de licencia

- **La Licencia Pública General de GNU (GPL)**. Esta es una llamada licencia "recíproca" que significa que si usa un software de código abierto con licencia bajo la licencia GPL, entonces debe hacer que el software sea de código abierto.
- **La licencia pública general menor de GNU (LGPL)** es una variante de la licencia GPL donde puede escribir componentes que se vinculan al código fuente abierto sin tener que publicar la fuente de estos componentes.
    - (“ Puedes usar componentes con licencia LGPL en tu software (incluso si tu software no es libre), sin tener que publicar el código fuente de tu parte. Si no haces cambios a la librería no tienes por qué publicar nada, pero si modificas algo de la librería tienes que publicar los cambios relativos a la misma. Es intermedio entre recíproca y no recíproca”)
- **La Licencia de Distribución Estándar de Berkley (BSD)**. Esta es una licencia no recíproca, lo que significa que no está obligado a volver a publicar ningún cambio o modificación realizada en el código fuente abierto. Puede incluir el código en los sistemas propietarios que se venden

Gestión de Licencias

- Establecer un sistema para mantener la información sobre los componentes de código abierto que se descargan y utilizan.
- Tener en cuenta los diferentes tipos de licencias y entender cómo se licencia un componente antes de usarlo.
- Estar atento a las vías de evolución de los componentes (puede que quede abandonado, deje de ser compatible, cambie la licencia…)
- Educar a la gente sobre el código abierto.
- Tener sistemas de auditoría en su lugar.
- Participar en la comunidad de código abierto (“No solo consumir, también contribuir y colaborar.”)

Tema 4. Pruebas

Pruebas de Programas: definición …

- El objetivo de las pruebas es mostrar que un programa hace lo que está destinado a hacer y descubrir los defectos del programa antes de que se ponga en uso.
- Cuando prueba el software, ejecuta un programa utilizando datos artificiales.
- Verifica los resultados de la ejecución de prueba para detectar errores, anomalías o información sobre los atributos no funcionales del programa.
- Puede revelarse la presencia de errores, NO de su Ausencia.
- Las pruebas forman parte de un proceso de verificación y validación más general, que también incluye técnicas de validación estática

Debug/depuración:

"La depuración (debugging) es el proceso de identificar, localizar y corregir los errores encontrados durante las pruebas u otras fases del desarrollo. Es una actividad posterior o complementaria a las pruebas. Primero se detecta el fallo con pruebas, y luego se corrige mediante depuración. Ejemplo: Tras una prueba que falla, el programador usa el depurador (debugger) para rastrear el error en el código y arreglarlo."

Fines de las Pruebas de Programas

1. **Demostrar al desarrollador y al cliente que el software cumple con sus requisitos.**
   - El primer objetivo lleva a las **pruebas de validación**.
   - Espera que el sistema funcione correctamente utilizando un conjunto determinado de casos de prueba que reflejen el uso esperado del sistema.
   - **Una prueba exitosa muestra que el sistema funciona según lo previsto.**
   - Para el **software personalizado**, esto significa que debe haber **al menos una prueba para cada requisito en el documento de requisitos**.
   - Para los productos de **software genéricos (**a público general, uso amplio), significa que debe haber **pruebas para todas las características del sistema, más combinaciones de estas características**, que se incorporarán en la versión del producto.

2. Para **descubrir situaciones en las que el comportamiento del software es incorrecto, indeseable o no se ajusta a su especificación.**
   - El segundo objetivo lleva a la **prueba de defectos.**
   - Los casos de prueba están diseñados para exponer defectos. Los casos de prueba en prueba de defectos pueden ser deliberadamente oscuros ("a joder") y no necesitan reflejar cómo se usa normalmente el sistema.
   - Una prueba exitosa es una prueba que hace que el sistema funcione incorrectamente y, por lo tanto, expone un defecto en el sistema

- Las pruebas de defectos están relacionadas con **eliminar el comportamiento no deseado del sistema,** como los bloqueos del sistema, las interacciones no deseadas con otros sistemas, los cálculos incorrectos y la corrupción de datos.++

Verificación vs. Validación

Validación

- Evaluación de un sistema o componente, durante o al final del proceso de desarrollo, para comprobar si se satisfacen los requisitos especificados.
- El software debe hacer lo que el usuario realmente requiere
- ¿Estamos construyendo el sistema correcto?

Verificación

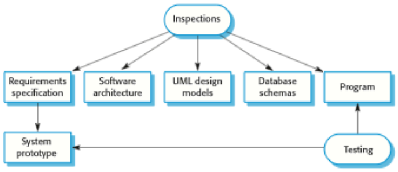
- Evaluación de un sistema o componente para determinar si los productos obtenidos satisfacen las condiciones impuestas. **Evaluación de la calidad de la implementación**
- El software debe ajustarse a su especificación
- ¿Estamos construyendo correctamente el sistema?

Necesidad de la V & V (validacion y verificacion)

- El objetivo de V & V es establecer la confianza de que el **sistema es "apto para su propósito"**. Depende del propósito del sistema, las expectativas del usuario y el entorno de marketing.
- **Propósito del software:** El nivel de confianza depende de cuán crítico sea el software para una organización (Cuanto más crítico es el sistema, más estricta debe ser la V&V)
- **Expectativas del usuario**: Los usuarios pueden tener bajas expectativas de ciertos tipos de software (Las expectativas determinan cuánto se invierte en V&V)
- **Entorno de marketing**: Obtener un producto en el mercado temprano puede ser más importante que encontrar defectos en el programa.

Inspección y Prueba

- **Inspecciones de software (1):** análisis de la representación estática del sistema para descubrir problemas (**verificación estática**) (“solo verif”)
    - Puede ser complementado por documento basado en herramientas y análisis de código.
    - “Se revisa el código, los documentos o modelos del sistema sin ejecutarlo, para encontrar errores, inconsistencias o malas prácticas.”
- **Pruebas de software (2):** enfocadas a la observación del comportamiento del producto (**verificación dinámica**) (“sirve para validación+verificación, depende tipo prueba”)
    - El sistema se ejecuta con datos de prueba y se observa su comportamiento operacional.



1. Inspecciones de Software

- Se trata de personas que examinan las representaciones del software con el objetivo de descubrir anomalías y defectos.
- Las inspecciones no requieren la ejecución de un sistema, por lo que se pueden usar antes de la implementación completa del software.
- Pueden aplicarse a cualquier representación del sistema (requisitos, diseño, datos de configuración, datos de prueba, etc.).
- Se ha demostrado que son una técnica eficaz para descubrir errores de programa.

Ventajas de la Inspección:

- Durante las pruebas, los errores pueden enmascarar (ocultar) otros errores. Debido a que la inspección es un proceso estático, no tiene que preocuparse por las interacciones entre los errores.
- Las versiones incompletas de un sistema se pueden inspeccionar sin costos adicionales. Si un programa está incompleto, no se necesita desarrollar artefactos de prueba especializados para probar las partes que están disponibles.
- Además de buscar defectos en los programas, una inspección también puede considerar atributos de calidad más amplios de un programa, como el cumplimiento de los estándares, la portabilidad y la capacidad de mantenimiento

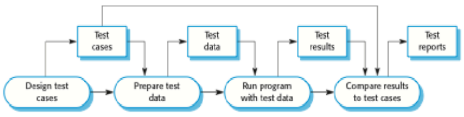
Las inspecciones **pueden verificar la conformidad con una especificación** pero no la conformidad con los requisitos reales del cliente, ni características no funcionales, como el rendimiento, la facilidad de uso, etc. (“verificación solo”).

2. Técnicas de prueba basadas en el código

- Partimos de que no puede hacerse una prueba exhaustiva (excepto en módulos triviales).
- **Pruebas de caja negra**
  - Basadas en el flujo de datos
  - Se usan cuando se quiere probar lo que hace el software pero no cómo lo hace
  - Comprobar que la salida es la esperada para cierta entrada
- **Pruebas de caja blanca**
  - Basadas en el flujo de control
  - Pruebas unitarias que se usan cuando se conoce la estructura interna y el funcionamiento del código a probar
  - Ej: en Java, cuando “miramos el código”; “ej: probar que una función if/else cubre todos los casos posibles”

    - Comprobar que bucles if… funcionen correctamente
- Ambas son dinámicas (es necesario ejecutar el software para detectar errores)

Modelo de proceso de pruebas de software



Etapas de las Pruebas

- Pruebas de desarrollo, donde el sistema se prueba durante el desarrollo para descubrir errores y defectos.
- Pruebas de lanzamiento, donde un **equipo de pruebas independiente** prueba una versión completa del sistema antes de que se lance a los usuarios.
- Pruebas de usuario, donde los usuarios o usuarios potenciales de un sistema prueban el sistema en su propio entorno.

1.Pruebas de Desarrollo

Las pruebas de desarrollo incluyen todas las actividades de prueba que lleva a cabo el equipo que desarrolla el sistema.

- **Prueba de unidad**, donde se prueban unidades de programa individuales o clases de objetos. Las pruebas unitarias deben centrarse en probar la **funcionalidad** de los objetos o métodos.
    - Tipo de prueba de defectos
    - Las unidades pueden ser:
      -**Funciones** o métodos individuales dentro de un objeto.
      -**Clases** de objetos con varios atributos y métodos.
      -**Componentes compuestos** con interfaces definidas utilizadas para acceder a su funcionalidad (pequeños; si mientras se puedan probar de forma aislada.)
- **Pruebas de componentes/integración**, donde **se integran varias unidades individuales** para crear componentes compuestos. Las pruebas de componentes deben centrarse en probar las **interfaces** de componentes.
    - Se comprueba que módulos ya probados en las pruebas unitarias de forma aislada, pueden interactuar correctamente
- **Pruebas del sistema**, donde algunos o todos los componentes de un sistema están integrados y el sistema se prueba como un todo. Las pruebas del sistema deben centrarse en probar las **interacciones** de los componentes.

Pruebas Unitarias

Pruebas unitarias de Objetos de Clases

- La cobertura completa del examen de una clase implica probar todas las operaciones asociadas con un objeto.
  - Interrogación de todos los atributos del objeto.
  - Probar el objeto en todos los estados posibles.
- La herencia hace que sea más difícil diseñar pruebas de clase de objeto, ya que la información que se va a probar no está localizada.

Pruebas unitarias Automáticas

- Siempre que sea posible, las pruebas unitarias deben automatizarse para que las pruebas se ejecuten y verifiquen sin intervención manual.
- En las **pruebas unitarias automatizadas**, utiliza un marco de automatización de pruebas (como JUnit, pytest) para escribir y ejecutar las pruebas del programa.
- Los marcos de prueba de unidad proporcionan clases de prueba genéricas que se extienden para crear casos de prueba específicos. Luego, pueden **ejecutar todas las pruebas que haya implementado e informar, a menudo a través de alguna GUI, sobre el éxito de las pruebas**
- Componentes de pruebas automáticas:
  - Una parte de configuración, donde inicializa el sistema con el caso de prueba, a saber, las entradas y las salidas esperadas.
  - Una parte de llamada, donde se llama al objeto o método a probar.
  - Una parte de aserción donde se compara el resultado de la llamada con el resultado esperado. Si la afirmación se evalúa como verdadera, la prueba ha sido exitosa. Si es falsa, entonces ha fallado.

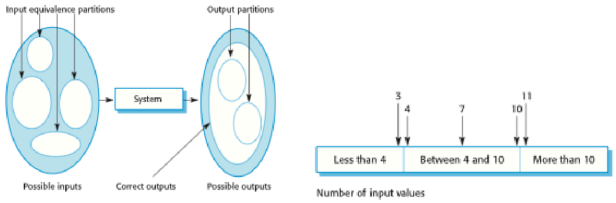
Efectividad de las pruebas unitarias

- Los casos de prueba deben mostrar que, cuando se usan como se espera, el componente que se está probando hace lo que se supone que debe hacer.
- Si hay defectos en el componente, estos deben ser revelados por los casos de prueba.
- Esto lleva a 2 tipos de caso de prueba unitaria:
  - El primero de ellos debe reflejar el funcionamiento normal de un programa y debe mostrar que el componente funciona como se espera.
  - El otro tipo de caso de prueba debe basarse en la experiencia de donde surgen problemas comunes. Debería usar entradas anormales para verificar que se procesen correctamente y no bloqueen la componente.

Tipos de pruebas unitarias

- **Pruebas de partición**, donde se identifican grupos de entradas que tienen características comunes y se deben procesar de la misma manera. Debe elegir las pruebas dentro de cada uno de estos grupos.
  - Los datos de entrada y los resultados de salida a menudo caen en clases diferentes donde todos los miembros de una clase están relacionados.
  - Cada una de estas clases es una **partición o dominio de equivalencia** (“grupo de valores de entrada para los que el programa se comporta de la misma forma”)

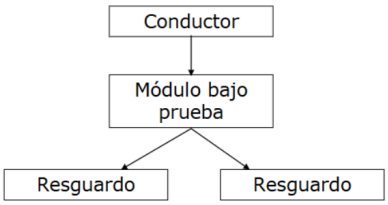
donde el programa se comporta de una manera equivalente para cada miembro de la clase. Los casos de prueba deben ser elegidos de cada partición.



- **Pruebas basadas en pautas**, donde se usan las pautas para elegir los casos de prueba. Estas pautas reflejan la experiencia previa de los tipos de errores que los programadores suelen cometer al desarrollar componentes.

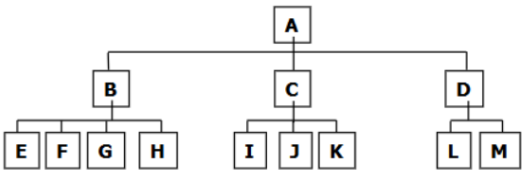
Pruebas de integración

- Aproximaciones:
  - Big-bang
  - Descendente (Top-down)
  - Ascendente (Bottom-up)
  - Sandwich: Combinación de las 2 anteriores.
- Pueden ser necesario escribir objetos simulados:
  - **Conductor**: simulan la llamada al módulo pasándole los parámetros necesarios para realizar la prueba
  - **Resguardos**: módulos simulados llamados por el módulo bajo prueba



1. Integración descendente (Top-down)

1. Probar el módulo A primero
2. Escribir módulos simulados, resguardos (stubs) para B, C, D
3. Una vez eliminados los problemas con A, probar el módulo B
4. Escribir resguardos para E, F, G, H etc.



2. Integración ascendente (Bottom-up)

1. Escribir conductores para proporcionar a E los datos que necesita de B

2. Probar E independientemente. Probar F independientemente, etc.
3. Una vez E, F, G, H han sido probados, el subsistema B puede ser probado, escribiendo un conductor para A

3. Big-Bang

Escribir y probar A, B, C, D, E, F, G, H, I, J, K, M a la vez.

4.Sandwich

Combinación de ascendente y descendente. “Se integran y prueban simultáneamente los módulos superiores (más abstractos, como la lógica de negocio) y los módulos inferiores (como acceso a datos), para luego unirlos en el medio.”

Pruebas de Interfaz (en integración)

- Los objetivos son **detectar fallos debidos a errores de interfaz o suposiciones no válidas sobre las interfaces** (“punto de contacto por el que dos partes del sistema se comunican o interactúan entre sí”).
    - Diseñar pruebas para que los **parámetros de un procedimiento llamado estén en los extremos de sus rangos.**
    - Diseñar pruebas que hacen que el **componente falle**. Utilizar las **pruebas de estrés en los sistemas de paso de mensajes.**
    - En los sistemas de memoria compartida, **variar el orden en que se activan los componentes.**
- Tipos de interfaz
    - **Interfaz de parámetros**: Datos pasados de un método o procedimiento a otro. (“comprobar que los datos enviados entre funciones/procedimientos coincidan con lo que espera el receptor.”)
    - **Interfaces de memoria compartida**: La memoria se comparte entre procedimientos o funciones. (“Las pruebas deben asegurar que se accede de forma segura y coherente, sin condic carrera…”)
    - **Interfaces de procedimientos:** El subsistema encapsula un conjunto de procedimientos a los que pueden llamar otros subsistemas. (“Las pruebas deben asegurar que las funciones disponibles se llaman correctamente, y que los datos que devuelven son correctos.”)
    - **Interfaces de paso de mensajes:** Los subsistemas solicitan servicios de otros subsistemas (“se prueban los mensajes enviados y recibidos, su contenido, formato…”)

Errores de Interfaz

- Mal uso de la interfaz: Un componente que llama a otro componente y comete un error en el uso de su interfaz, por ejemplo entregar parámetros en un orden incorrecto.
- Malentendido interfaz: Un componente que llama incorpora suposiciones sobre el comportamiento del componente llamado que son incorrectos.
- Errores de tiempo: El componente llamado y el componente de llamada operan a diferentes velocidades y se accede a la información desactualizada

Pruebas de sistema

- Las pruebas del sistema durante el desarrollo implican **la integración de componentes para crear una versión del sistema y luego probar el sistema integrado.**
- El enfoque en las pruebas del sistema es **probar las interacciones entre los componentes.**
    - Las pruebas del sistema **comprueban que los componentes son compatibles, interactúan correctamente y transfieren los datos correctos en el momento correcto a través de sus interfaces.**
- Durante la prueba del sistema, los componentes reutilizables que se han desarrollado por separado y los sistemas estándar pueden integrarse con los componentes desarrollados recientemente. Luego se prueba el sistema completo.
- Los componentes desarrollados por diferentes miembros del equipo o sub-equipos pueden integrarse en esta etapa. La prueba del sistema es un proceso colectivo más que individual.
- En algunas empresas, las pruebas del sistema pueden involucrar a un **equipo de pruebas independiente** sin la participación de diseñadores y programadores

Pruebas con casos de uso (de sist)

- Los casos de uso desarrollados para identificar las interacciones del sistema se pueden utilizar como base para las pruebas del sistema.
- Cada caso de uso usualmente involucra varios componentes del sistema, por lo que probar el caso de uso obliga a que estas interacciones ocurran.
- Los diagramas de secuencia asociados con el caso de uso documentan los componentes e interacciones que se están probando.

Estrategia de Pruebas

- La prueba exhaustiva del sistema es imposible, por lo que se deben desarrollar políticas de prueba que definan la cobertura de prueba requerida del sistema.
- (“Los casos de uso te dicen qué funcionalidades reales probar. La estrategia de pruebas te dice cómo organizar esas pruebas, cuánto cubrir y con qué criterios”)
- Ejemplos de políticas de prueba:
    - Se deben probar todas las funciones del sistema a las que se accede a través de los menús.
    - Se deben probar las combinaciones de funciones a las que se accede a través del mismo menú.
    - Todas las funciones deben probarse con la entrada correcta e incorrecta.

Pruebas de lanzamiento (release) (tipo de prueba del sistema; prueba de validación)

- **Un equipo separado que no haya estado involucrado en el desarrollo del sistema, debe ser responsable de las pruebas de lanzamiento.**
- La prueba de lanzamiento es el proceso de **probar un lanzamiento particular de un sistema** que está diseñado para ser usado fuera del equipo de desarrollo.
- El objetivo principal del proceso de prueba de lanzamiento es convencer al proveedor del sistema de que **el sistema cumple con sus requisitos y es lo suficientemente bueno para uso externo (pruebas de validación)**
    - VS pruebas normales del sistema hechas por equipo de desarrollo: centrarse en descubrir errores en el sistema (pruebas de defectos)
    - Por lo tanto, las pruebas de lanzamiento tienen que mostrar que el sistema entrega su funcionalidad, rendimiento y confiabilidad específicos, y que no falla durante el uso normal.
- La prueba de lanzamiento generalmente es un **proceso de prueba de caja negra** donde las pruebas solo se derivan de las especificaciones del sistema

Pruebas de prestaciones/rendimiento (caja negra)

- **Parte de las pruebas de lanzamiento** puede implicar probar las propiedades emergentes de un sistema, como el rendimiento y la confiabilidad (velocidad de respuesta, uso de recursos…).
- Las pruebas deben **reflejar el perfil de uso del sistema.**
- Las pruebas de rendimiento generalmente implican la planificación de una serie de pruebas en las que **la carga aumenta constantemente hasta que el rendimiento del sistema se vuelve inaceptable**.
- Las **pruebas de estrés** son una forma de pruebas de rendimiento en las que el sistema se sobrecarga deliberadamente para probar su comportamiento hasta que falla. (“Ver si el sistema colapsa con elegancia (no pierde datos, no daña servicios). Identificar puntos críticos o cuellos de botella.”)

Pruebas de regresión

- Las pruebas de regresión prueban el sistema para verificar que los cambios no hayan "roto" el código que funcionaba anteriormente.
- En un proceso de prueba manual, **la prueba de regresión es costosa** pero, con la prueba **automatizada, es simple y directa**. Todas las pruebas se vuelven a ejecutar cada vez que se realiza un cambio en el programa.
- Las pruebas deben ejecutarse con éxito antes de que se confirme el cambio

Prueba de usuario (prueba de validación. Caja negra)

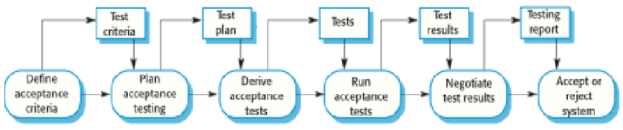
- La prueba del usuario o cliente es una etapa en el proceso de prueba en la que lo**s usuarios o clientes proporcionan información y consejos sobre la prueba del sistema.**
- La prueba del usuario es esencial, incluso cuando se han llevado a cabo pruebas exhaustivas del sistema y de la versión.

- La razón de esto es que las influencias del entorno de trabajo del usuario tienen un efecto importante en la confiabilidad, el rendimiento, la facilidad de uso y la solidez de un sistema. Estos no pueden ser replicados en un entorno de prueba.

Tipos de prueba de usuario:

- **Prueba alfa:** Los usuarios del software trabajan con el equipo de desarrollo para probar el software en el sitio del desarrollador.
- **Prueba beta:** Un lanzamiento del software se pone a disposición de los usuarios para permitirles experimentar y plantear los problemas que descubren a los desarrolladores del sistema.
- **Test de aceptación:** Los clientes prueban un sistema para decidir si está listo o no para ser aceptado por los desarrolladores del sistema y desplegado en el entorno del cliente. Principalmente para sistemas personalizados. Etapas:
    - Definir criterios de aceptación: ¿Qué condiciones debe cumplir el sistema para que el cliente lo considere aceptable?
    - Plan de pruebas de aceptación: documento que describe cómo se va a realizar el test de aceptación
    - Derivar pruebas de aceptación: Se diseñan casos de prueba concretos a partir de los criterios.
    - Ejecutar pruebas de aceptación: El cliente o usuario final realiza las pruebas en un entorno real o de prueba que simula el entorno de producción
    - Negociar los resultados de la prueba: Cliente y desarrollador se reúnen para analizar los resultados.
    - Rechazar / aceptar sistema

The acceptance testing process



RESUMEN: Puntos Clave

Las pruebas solo pueden mostrar la presencia de errores en un programa. No puede demostrar que no hay fallos.

Las pruebas de desarrollo son responsabilidad del equipo de desarrollo de software. Un equipo independiente debe ser responsable de probar un sistema antes de que sea entregado a los clientes.

Las pruebas de desarrollo incluyen pruebas unitarias, en las que prueba los objetos individuales, los métodos de prueba de componentes en los que prueba grupos relacionados de objetos, y pruebas de sistemas, en los que prueba sistemas parciales o completos

“Un oráculo es un mecanismo que nos dice cuál es el resultado esperado de una prueba, y nos permite compararlo con el resultado real obtenido al ejecutar el software.”

—----------

EJERCICIOS

Complejidad ciclomática de McCabe

Es una métrica basada en la estructura de control del código, proporciona una medida cuantitativa para probar la dificultad y una indicación de la fiabilidad última. También puede emplearse para proporcionar una indicación del tamaño máximo del módulo.

La complejidad ciclomática de McCabe se puede calcular de las tres formas siguientes:

1- V(G)=a-n+2 siendo a el número de arcos o aristas del grafo y n el número de nodos.
2- V(G)=r siendo r el número de regiones cerradas del grafo.
3- V(G)=c+1 siendo c el número de nodos de condición.

De cualquiera de los tres modos el valor a obtener debe ser el mismo

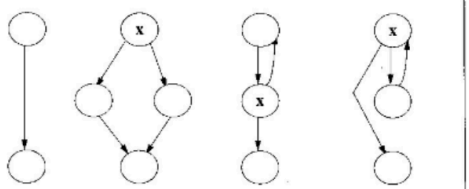
Como se puede apreciar para poder realizar el cálculo hemos tenido que obtener un grafo a partir del código del programa. Un grafo es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto. Un grafo se representa gráficamente como un conjunto de puntos (vértices o nodos) unidos por líneas (aristas). Desde un punto de vista práctico, los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras. Por ejemplo, una red de ordenadores puede representarse y estudiarse mediante un grafo, en el cual los vértices representan equipos y las aristas representan conexiones

Cómo dibujar un grafo de un código:

Para dibujar el grafo de flujo de un programa es recomendable seguir los siguientes pasos:

1. Señalar sobre el código cada condición de cada decisión (por ejemplo, «mismo producto» en la figura 12-4 es una condición de la decisión «haya registros y mismo producto») tanto en sentencias If-then y Case of como en los bucles While o Until.

2. Agrupar el resto de las sentencias en secuencias situadas entre cada dos condiciones según los esquemas de representación de las estructuras básicas que mostramos a continuación.



Secuencia | Si x entonces... (If x then...else...) | Hacer... hasta x (Do...until x) | Mientras x hacer... (While x do...)

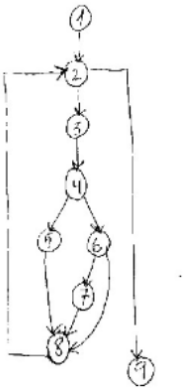
3. Numerar tanto las condiciones como los grupos de sentencias, de manera que se les asigne un identificador único. Es recomendable alterar el orden en el que aparecen las condiciones en una decisión multicondicional, situándolas en orden decreciente de restricción (primero, las más restrictivas). El objetivo de esta alteración es facilitar la derivación de casos de prueba una vez obtenido el grafo. Es conveniente identificar los nodos que representan condiciones asignándoles una letra y señalar cuál es el resultado que provoca la ejecución de cada una de las aristas que surgen de ellos (por ejemplo, si la condición x es verdadera o falsa).

Ej:

```
Ejemplo2
…….
tamaño=25;
mayor=menor=vector[0];
i=1;
while(i<tamaño){
        total=+vector[i];
        if(vector[i]>mayor) mayor=vector[i];
        else if(vector[i]<menor) menor=vector[i];
        i++;
}
System.out.println(mayor);
System.out.println(menor);
```

SOLUCIÓN

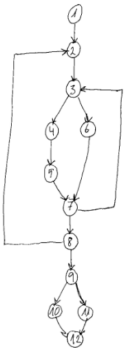
```
tamaño=25;
mayor = menor = vector[0];
i = 1;
while (i<tamaño){
    total = +vector[i];
    if (vector[i] > mayor)
        mayor = vector[i];
    else if (vector[i] < menor)
        menor = vector[i];
    i++;
System.out.println(mayor);
System.out.println(menor);
```



```
Ejemplo3
pares=impares=0;
for(i=1;i<11;i++){
        for(j=1;j<11;j++){
                if((i*j)%2==0){
                        System.out.println(i*j) ;
                        pares++;
                }
                else impares++;
        }
}
if(pares>impares) System.out.println(pares);
else System.out.println(impares);
```

SOLUCIÓN

```
pares=impares=0
for(i=1; i<11; i++){
  for(j=1;j<11;j++){
    if((i*j)%2==0){
      System.out.println(i*j);
      pares++;
    }
    else impares++;
  }
}
if (pares>impares)
    System.out.println(pares);
else System.out.println(impares);
```



Método del camino básico

Además de medir la complejidad del código, v(g), se utiliza para pruebas de código. Se trata de un tipo de **prueba de caja blanca** donde se supone como cierto que, para todo código expresado en forma de grafo, es posible definir un camino básico susceptible de ser sometido a pruebas rigurosas. El método tiene cuatro pasos:

1. Elaborar el grafo de flujo asociado al programa a partir del diseño (o del código fuente) del mismo.
2. Calcular la complejidad ciclomática v(g), estableciendo el número de caminos independientes. Se denomina independiente a cualquier camino que introduce al menos una sentencia de procesamiento (o condición) no considerada anteriormente.
3. Seleccionar un conjunto de caminos básicos.
4. Generar casos de prueba para cada uno de los caminos del paso anterior, seleccionando las entradas adecuadas para asegurar que todos se prueben

Clases de equivalencia

Un problema importante en las **pruebas de caja negra** es la incertidumbre sobre la cobertura de las mismas. Generalmente, la amplia casuística de los valores de entrada obliga a utilizar técnicas que permitan racionalizar el número de pruebas a realizar. La técnica de las clases de

equivalencia permite reducir esta casuística de manera considerable. Las clases de equivalencia consisten en **identificar un conjunto finito de datos que será suficiente para realizar las pruebas cubriendo aceptablemente los distintos casos posibles.**

Diagramas de clases

- **+** = pública
- **-** = privada
- **#** = protegida

TEMA 5: mantenimiento

- El software no se deteriora con el uso ni con el tiempo.
- Deterioro en su estructura al introducir cambios (erosión del diseño).
- Cambios en el software para hacerlo más mantenible.

Tipos de mantenimiento de un sw:

ISO/IEC 14764

	Corrección	**Mejora**
Proactiva	*Preventivo*	*Perfectivo*
Reactiva	*Correctivo*	*Adaptativo*

1. **Correctivo** (reactivo + corrección): Se realiza después de que se descubre un error o fallo.
2. **Preventivo** (proactivo + corrección): Se realiza antes de que aparezcan errores, anticipando problemas. Mejorar la estructura interna del software para evitar errores futuros
3. **Adaptativo** (reactivo + mejora): Se realiza para adaptar el software a cambios en su entorno (hardware, SO, normativas, etc.). EJ: Actualizar el software para que funcione en una nueva versión de Windows.
4. **Perfectivo** (proactivo + mejora): Se hace para mejorar el rendimiento, eficiencia, usabilidad o mantenibilidad del sistema , o añadir nuevas funcionalidades, sin que haya un fallo.

EVOLUCIÓN

El cambio de software es inevitable:

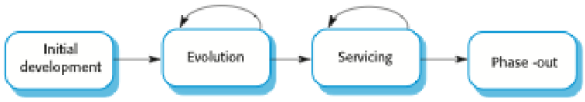
- Surgen nuevos requisitos cuando se utiliza el software;
- El entorno empresarial cambia;
- Los errores deben ser reparados;
- Se agregan nuevas computadoras y equipos al sistema;
- El rendimiento o la confiabilidad del sistema debe ser mejorado.

Un problema clave para todas las organizaciones es implementar y administrar cambios en sus sistemas de software existentes.

Importancia de la evolución

- Las organizaciones tienen grandes inversiones en sus sistemas de software: son activos empresariales críticos.
- Para mantener el valor de estos activos para el negocio, se deben cambiar y actualizar.
- La mayoría del presupuesto de software en las grandes empresas se dedica a cambiar y evolucionar el software existente en lugar de desarrollar un nuevo software

Evolución y fases de un Software

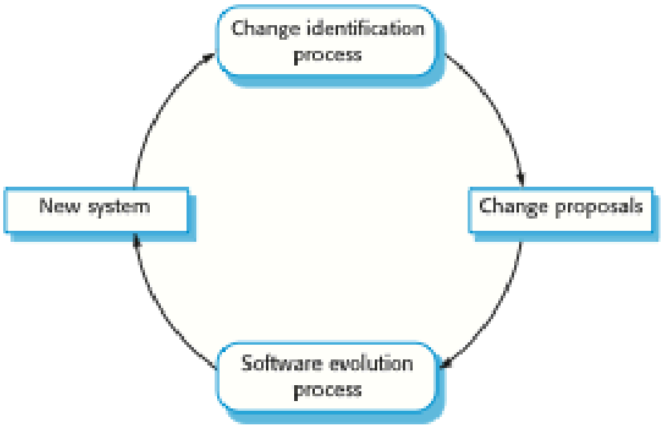


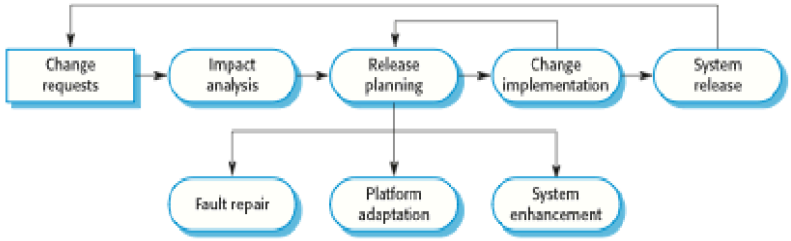
- Evolución
  - La etapa en el ciclo de vida de un sistema de software donde está en uso operacional y está evolucionando a medida que se proponen e implementan **nuevos requisitos** en el sistema.
- Servicio
  - En esta etapa, el software sigue siendo útil, pero los únicos cambios realizados son los necesarios para mantenerlo operativo, es decir, las **correcciones de errores** y los cambios para reflejar los **cambios en el entorno** del software.
  - No se agrega ninguna nueva funcionalidad.
- Reducir progresivamente/"phase out"
  - El software puede seguir utilizándose, pero no se le hacen más cambios

Proceso de evolución

- Los procesos de evolución del software dependen de
  - El tipo de software que se mantiene;
  - Los procesos de desarrollo utilizados; (metodología ágil, en cascada…)
  - Las habilidades y experiencia de las personas involucradas.
- Las propuestas de cambio son el motor de la evolución del sistema.
  - Debe vincularse con los componentes que se ven afectados por el cambio, lo que permite estimar el costo y el impacto del cambio.
- La identificación y evolución del cambio continúa durante toda la vida útil del sistema

Identificación de cambios y proceso de evolución:

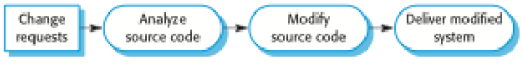




Implementación de cambios (fase del proceso de evolucion)

- Iteración del proceso de desarrollo donde se diseñan, implementan y prueban las revisiones del sistema.
- Una diferencia crítica con el desarrollo desde cero es que la primera etapa de la implementación del cambio **puede involucrar la comprensión del programa**, especialmente si los desarrolladores del sistema original no son responsables de la implementación del cambio.
- Durante la **fase de comprensión del programa**, debe comprender cómo está estructurado el programa, cómo proporciona funcionalidad y cómo el cambio propuesto podría afectar al programa.

Requerimientos de cambios urgentes



Es posible que se deban implementar cambios urgentes sin pasar por todas las etapas del proceso de ingeniería del software.

- Si es necesario reparar un fallo grave del sistema para permitir que el funcionamiento normal continúe;
- Si los cambios en el entorno del sistema (por ejemplo, una actualización del sistema operativo) tienen efectos inesperados;
- Si hay cambios en el negocio que requieren una respuesta muy rápida (por ejemplo, el lanzamiento de un producto de la competencia)

Evolución de métodos ágiles

- Los métodos ágiles se basan en el desarrollo incremental, por lo que la transición del desarrollo a la evolución es perfecta. (“En metodologías ágiles se entrega el sistema poco a poco en versiones funcionales (releases o sprints), por eso, no hay una separación clara entre "desarrollo" y "mantenimiento/evolución" “)
- La evolución es simplemente una **continuación del proceso de desarrollo basado en lanzamientos frecuentes del sistema**.
- Las **pruebas de regresión automatizadas** (“asegurar que lo que ya funcionaba sigue funcionando.”) son particularmente valiosas cuando se realizan cambios en un sistema.
- Los cambios pueden expresarse como historias de usuario adicionales (“En los métodos ágiles, todo lo que el sistema debe hacer se describe como historias de usuario”)

Problemas de “Handover”

- Cuando el equipo de desarrollo ha utilizado un enfoque ágil, pero el equipo de evolución no está familiarizado con los métodos ágiles y prefiere un enfoque basado en el plan.
    - El equipo de evolución puede esperar documentación detallada para respaldar la evolución y esto no se produce en procesos ágiles.
- Cuando se ha utilizado un enfoque basado en planes para el desarrollo, pero el equipo de evolución prefiere utilizar métodos ágiles.
    - El equipo de evolución puede tener que comenzar desde cero a desarrollar pruebas automatizadas si no las incluyeron y es posible que el código del sistema no se haya refactorizado y simplificado como se espera en el desarrollo ágil.

Dinámica de la evolución de programas

- La dinámica de la evolución del programa es **el estudio de los procesos de cambio de sistema.**
- Después de varios estudios empíricos importantes, **Lehman y Belady** propusieron que había **una serie de "leyes" que se aplicaban a todos los sistemas a medida que evolucionaban.**
- Son observaciones empíricas/sensatas más que leyes. Son aplicables a grandes sistemas desarrollados por grandes organizaciones.
- No está claro si estos son aplicables a otros tipos de sistemas de software.

El cambio es inevitable

“Antes de las leyes, Lehman observó algo esencial: Los requisitos cambian incluso durante el desarrollo, porque el entorno cambia."

- Es probable que los requisitos del sistema cambien mientras se desarrolla el sistema porque el entorno está cambiando. Por lo tanto, un sistema entregado no cumplirá con sus requisitos!
- Los sistemas están estrechamente acoplados con su entorno. Cuando un sistema se instala en un entorno, cambia ese entorno y, por lo tanto, cambia los requisitos del sistema.
- Los sistemas DEBEN ser cambiados para que sigan siendo útiles en un entorno.

Leyes de Lehman

- Cambio continuo: Un programa que se utiliza en un entorno del mundo real debe cambiar necesariamente o, de lo contrario, ser cada vez menos útil en ese entorno.
- Incremento de la complejidad: a medida que un programa en evolución cambia, su estructura tiende a hacerse más compleja. Se deben dedicar recursos adicionales para preservar y simplificar la estructura.
- Autoregulación: La evolución del programa es un proceso de autorregulación. Los atributos del sistema, como el tamaño, el tiempo entre versiones y el número de errores informados son aproximadamente invariantes para cada versión del sistema.
- Estabilidad organizativa : A lo largo de la vida útil de un programa, su tasa de desarrollo es aproximadamente constante e independiente de los recursos dedicados al desarrollo del sistema
- Conservación de la familiaridad: Durante la vida útil de un sistema, el cambio incremental en cada versión es aproximadamente constante.
- Crecimiento continuo: La funcionalidad ofrecida por los sistemas tiene que aumentar continuamente para mantener la satisfacción del usuario.
- Disminución de calidad: La calidad de los sistemas disminuirá a menos que se modifiquen para reflejar los cambios en su entorno operativo.

Aplicabilidad de las leyes de Lehman

- Las leyes de Lehman parecen ser generalmente aplicables a **grandes sistemas desarrollados por grandes organizaciones.**
- Confirmado a principios de 2000 por el trabajo de Lehman en el proyecto FEAST.
- No está claro cómo se deben modificar para pequeñas organizaciones o sistemas de tamaño medio.

Puntos Clave: RESUMEN

- Para sistemas personalizados, **los costos de mantenimiento del software generalmente exceden los costos de desarrollo del software**.

- El proceso de evolución del software está impulsado por solicitudes de cambios e incluye el análisis del impacto del cambio, la planificación de la versión y la implementación del cambio.
- Las leyes de Lehman, como la noción de que el cambio es continuo, describen una serie de ideas derivadas de estudios a largo plazo de la evolución del sistema.

MANTENIMIENTO

- Modificar un programa después de que haya sido puesto en uso. (parte de la evolucion)
- El término se utiliza principalmente para cambiar el software personalizado. Se dice que los productos de software genéricos evolucionan para crear nuevas versiones.
- El mantenimiento normalmente no implica cambios importantes en la arquitectura del sistema.
- Los cambios se implementan modificando los componentes existentes y agregando nuevos componentes al sistema.

MS consume muchos recursos.

Actividades de MS:

- Comprender software.
- Modificar software y actualizar documentación.
- Realización de pruebas.

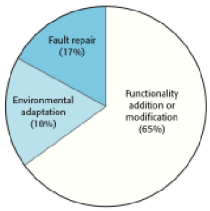
Costes ocultos:

- Se pierden oportunidades de desarrollo (“no puedes hacer nuevas cosas si estas ocupado manteniendo otro”).
- Insatisfacción del cliente (“Si los errores se corrigen lento o el sistema funciona mal, el cliente pierde confianza.”)
- Errores ocultos introducidos en el mantenimiento (“Al hacer cambios, se pueden romper otras partes sin darnos cuenta.”)
- Perjuicio en otros proyectos. (“descuidar otros proyectos…”)

Tipos de mantenimiento

- Mantenimiento para **reparar fallos de software (correctivo).**
  - Cambiar un sistema para corregir deficiencias en la forma en que cumpla con sus requisitos.
- Mantenimiento para **adaptar el software a un entorno operativo diferente (adaptativo)**.
  - Cambiar un sistema para que funcione en un entorno diferente (computadora, sistema operativo, etc.) desde su implementación inicial
- Mantenimiento para **agregar o modificar la funcionalidad del sistema (perfectivo).**
  - Modificando el sistema para satisfacer nuevos requerimientos..
- (Mantenimiento preventivo: Cambios hechos para prevenir errores futuros, antes de que ocurran.)

Distribución de los esfuerzos de mantenimiento



Costes de mantenimiento

- Por lo general, mayor que los costos de desarrollo
- Afectados por factores tanto técnicos como no técnicos
- Aumenta a medida que se mantiene el software. El mantenimiento corrompe la estructura del software, lo que dificulta aún más el mantenimiento.
- El software de envejecimiento (“Se refiere a sistemas antiguos que usan tecnologías o lenguajes obsoletos, difíciles de mantener”) puede tener altos costos de soporte (por ejemplo, lenguajes antiguos, compiladores, etc.).

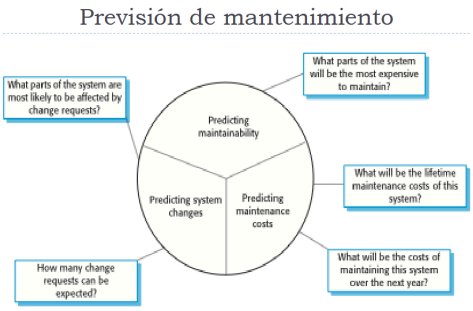
Aspectos de los costes de mantenimiento

- Estabilidad del equipo
  - Los costes de mantenimiento se reducen si el mismo personal está involucrado con ellos durante algún tiempo.
- Responsabilidad contractual
  - Los desarrolladores de un sistema pueden no tener responsabilidad contractual por el mantenimiento, por lo que no hay incentivos para diseñar para futuros cambios. (“Muchos equipos entregan un sistema y se desentienden. Si no tienen que mantenerlo, pueden escribir código difícil de entender o modificar sin preocuparse”)
- Habilidades del personal
  - El personal de mantenimiento a menudo no tiene experiencia y tiene un conocimiento limitado del dominio.
- Edad del programa y degradación de su estructura
  - A medida que los programas envejecen, su estructura se degrada y se vuelven más difíciles de entender y cambiar.

Previsión de mantenimiento

- La previsión de mantenimiento se refiere a la **evaluación de qué partes del sistema pueden causar problemas y tener altos costos de mantenimiento**
- La aceptación del cambio depende de la capacidad de mantenimiento de los componentes afectados por el cambio; (“Si un componente es difícil de entender o modificar, es más probable que el equipo evite hacerle cambios, incluso si el cliente lo pide.”)
- La implementación de cambios degrada el sistema y reduce su mantenibilidad;

- Los costos de mantenimiento dependen de la cantidad y facilidad de los cambios (los costos de cambio dependen de la capacidad de mantenimiento)



Previsión de cambios

- "Es el intento de anticipar cuántos y qué tan frecuentes serán los cambios que necesitará un sistema durante su vida útil"
- La predicción de la cantidad de cambios requiere de la comprensión de las relaciones entre un sistema y su entorno.
- Los sistemas estrechamente acoplados requieren cambios cada vez que se cambia el entorno. "Cuanto más estrechamente conectado esté un sistema con su entorno, más afectado se verá cuando ese entorno cambie."
- Los factores que influyen en esta relación son
    - Número y complejidad de las interfaces del sistema ("más dependencias= más posibilidades de necesitar cambios");
    - Número de requisitos del sistema inherentemente volátiles;
    - Los procesos de negocio donde se utiliza el sistema

Técnicas para el mantenimiento del software

Ingeniería inversa, reingeniería y reestructuración como solución: Reconstruir el software, ya sea reprogramándolo, redocumentándolo, rediseñándolo, o rehaciendo alguna/s característica/s del producto

Ingeniería inversa

La ingeniería inversa consiste en **analizar un sistema ya existente** para **identificar sus componentes y las relaciones entre ellos** y comprender cómo funciona, así como para **crear representaciones del mismo** en alguna forma que no exista, generalmente en un **nivel de abstracción más elevado (**"Reconstruir documentación, modelos o diagramas")
Hay tres tipos: (i)datos, (ii)procesos, (iii) interfaces .

(ii)Ingeniería inversa de procesos

Ingeniería inversa que se aplica sobre código de un programa para averiguar su lógica o sobre cualquier documento de diseño para obtener documentos de análisis o de requisitos. Pasos a seguir:

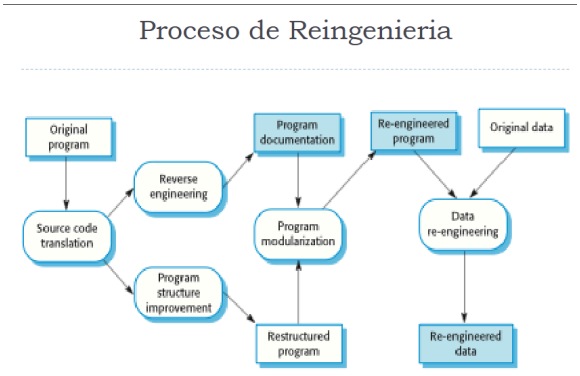
1. Buscamos el programa principal.
2. Ignoramos inicializaciones de variables, etc.
3. Inspeccionamos la primera rutina llamada y la examinamos si es importante.
4. Inspeccionamos las rutinas llamadas por la primera rutina del programa principal, y examinamos aquéllas que nos parecen importantes.
5. Repetimos los pasos 3-4 a lo largo del resto del software.
6. Recopilamos esas rutinas “importantes”, que se llaman componentes funcionales.
7. Asignamos significado a cada componente funcional:
    - Explicamos qué hace cada componente funcional en el conjunto del sistema y
    - Explicamos qué hace el sistema a partir de los diferentes componentes funcionales

Reingeniería

- Modificación de un producto 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.**
- “Necesaria” tras la degradación del sistemas por cambios y actualizaciones.
- **Reestructurar o reescribir parte o todo un sistema heredado sin cambiar su funcionalidad.**
- Aplicable cuando algunos, pero no todos los subsistemas de un sistema más grande, requieren un mantenimiento frecuente.
- La reingeniería implica un esfuerzo adicional para hacerlos más fáciles de mantener. El sistema puede ser re-estructurado y re-documentado.

Ventajas de la reingeniería

- Riesgo reducido. Existe un alto riesgo en el desarrollo de software nuevo: puede haber problemas de desarrollo, problemas de personal y problemas de especificaciones.
- Costo reducido: El costo de la reingeniería a menudo es significativamente menor que el costo de desarrollar un nuevo software



- Traducción de código fuente: Convertir código a un nuevo idioma. (“no tiene por qué ser lo primero ni hacerse siempre”. “Se convierte el código a otro lenguaje de programación más moderno o adecuado”)
- Ingeniería inversa: Analiza el programa para entenderlo.
- Mejora de la estructura del programa: Reestructura automáticamente para la comprensibilidad. “eliminar código duplicado, mejorar nombres de funciones, limpiar código innecesario.”
- Modularización del programa: Reorganizar la estructura del programa.
- Reingeniería de datos: Limpieza y reestructuración de datos del sistema. (“modificar estructuras de datos, normalizar bbdd…)

Factores que afectan al coste de la reingeniería

- La calidad del software para ser rediseñado.
- La herramienta de soporte disponible para reingeniería (“Contar con herramientas automáticas de análisis, documentación, traducción de código, etc., reduce el esfuerzo manual”)
- El alcance de la conversión de datos que se requiere. (“Si hay muchos datos, están mal organizados…, convertirlos al nuevo sistema puede ser costoso”)
- La disponibilidad de personal experto para la reingeniería. (“que conozcan el lenguaje original…”)
    - Esto puede ser un problema con los sistemas antiguos basados en tecnología que ya no se usa ampliamente.

Reingeniería : procesos implicados

Proceso:

- Análisis de inventario (“Se evalúan todas las aplicaciones existentes para decidir cuáles vale la pena reingenierizar”): antigüedad, importancia de la aplicación en el negocio, mantenibilidad actual.
- Reestructuración de documentos (“Se decide qué documentación falta o debe mejorarse y se lleva a cabo”): qué se va a documentar
- Ingeniería inversa. Almacenamiento en el repositorio
- Reestructuración o ingeniería directa. Decidir si:
    - Comprar un nuevo software
    - Desarrollar un nuevo software

Valoraciones de la reingeniería

Beneficio de mantener el software como está:

BM = [VA – (CMA + COpA)] x T.Vida

-BM: el beneficio de mantenimiento (Beneficio por mantener el software tal como está)
-VA: los ingresos del negocio actual (beneficio anual que genera el software)
-CMA: el coste de mantenimiento anual actual
-COpA: el coste anual actual de operación de la aplicación, es decir, los costes derivados de mantener la aplicación en uso (servicios de atención al cliente, administración,...).

-T.Vida: Años que se espera seguir usando el software

Beneficio reingeniería:

BR = [(GF x T.Vida) – (CR x FR)] – BM

GF = VF – (CMF + COpF)]

T.Vida = T.Vida Estimado – T. que tarda la Reingeniería

-BR: beneficio de reingeniería
-GF: ganancia final anual
-CR: coste de reingeniería
-FR: factor de riesgo de la reingeniería
-BM: beneficio de mantenimiento (se resta ya que ya se conseguiría sin reingeniería)
-VF: el valor de negocio tras la reingeniería (beneficio anual que generará el software)
-CMF: coste de mantenimiento anual final
-COpF: coste de operación anual final

Mantenimiento preventive mediante “refactoring”/refactorización

- La refactorización es el proceso de **realizar mejoras en un programa para reducir la degradación a través del cambio.**
- Puede pensar en la refactorización como un **"mantenimiento preventivo"** que reduce los problemas de cambios futuros.
- La refactorización implica modificar un programa para mejorar su estructura, reducir su complejidad o facilitar su comprensión.
- Cuando se refactoriza un programa, no debe agregar funcionalidad, sino concentrarse en la mejora del programa

“Refactoring” y reingeniería

- La reingeniería se lleva a cabo después de que un sistema se ha mantenido durante algún tiempo y los costos de mantenimiento están aumentando. Utiliza herramientas automatizadas para procesar y rediseñar un sistema heredado para crear un nuevo sistema que sea más fácil de mantener.
- La refactorización es un proceso continuo de mejora a lo largo del proceso de desarrollo y evolución. Está pensado para evitar la degradación de la estructura y el código que aumenta los costos y las dificultades de mantener un sistema.

Reestructuración

- Modificación del producto software dentro del mismo nivel de abstracción (“por ejemplo: nivel de código, sin rediseñar toda la arquitectura”).
- “La reestructuración es el proceso de modificar un producto software manteniéndose en el mismo nivel de abstracción, generalmente a nivel de código, sin cambiar su funcionalidad ni rediseñar toda la arquitectura del sistema”
- No altera el resultado de etapas anteriores o posteriores.
- No existe un conocimiento total del sistema, únicamente de la parte a modificar.

- **Refactoring** es un tipo de reestructuración de código, que se aplica en OO y consiste en cambiar el software de un sistema para mejorar su estructuración Interna pero sin alterar su comportamiento externo. “El refactoring es el proceso de modificar la estructura interna del código sin cambiar su comportamiento externo, con el objetivo de mejorar su legibilidad, organización, mantenibilidad y reducir su complejidad.”

‘Malas prácticas” programando

- Código duplicado
  - El mismo código o uno muy similar puede incluirse en diferentes lugares de un programa. Esto se puede eliminar e implementar como un solo método o función que se llama según sea necesario.
- Metodos largos
  - Si un método es demasiado largo, debe ser rediseñado como un número de métodos más cortos.
- Sentencias “Switch (case)”
  - Estos a menudo implican duplicación, donde el cambio depende del tipo de valor. Las instrucciones de cambio pueden estar dispersas alrededor de un programa. En los lenguajes orientados a objetos, a menudo se puede usar el polimorfismo para lograr lo mismo.
- Agrupamiento de datos
  - Los grupos de datos se producen cuando el mismo grupo de elementos de datos (campos en clases, parámetros en métodos) se repiten en varios lugares de un programa. A menudo estos se pueden reemplazar con un objeto que encapsula todos los datos.
- Generalidad especulativa
  - Esto ocurre cuando los desarrolladores incluyen generalidad en un programa en caso de que se requiera en el futuro. A menudo esto puede simplemente ser eliminado.

Facilidad de Mantenimiento

La facilidad de mantenimiento es la **disposición de un sistema o componente software para ser modificado** con objeto de corregir fallos, mejorar su funcionamiento u otros atributos, o adaptarse a cambios en el entorno

Factores que afectan a la facilidad de mantenimiento :

- Falta de cuidado en las fases de diseño, codificación o prueba.
- Pobre configuración del producto software.
- Inadecuada cualificación del equipo de desarrolladores del software.
- Estructura del software fácil de comprender.
- Facilidad de uso del sistema.
- Empleo de lenguajes de programación y sistemas operativos estandarizados.

- Estructura estandarizada de la documentación.
- Documentación disponible de los casos de prueba.
- Incorporación en el sistema de facilidades de depuración.
- Disponibilidad del equipo (computador y periféricos) adecuado para realizar el mantenimiento.
- Disponibilidad de la persona o grupo que desarrolló originalmente el software.
- Planificación del mantenimiento.

Tratamiento de sistemas "heredados"

Las organizaciones que dependen de sistemas heredados deben elegir una estrategia para la evolución de estos sistemas:

- Desechar completamente el sistema y modificar los procesos de negocio para que ya no sean necesarios;
- Continuar manteniendo el sistema;
- Transformar el sistema mediante reingeniería para mejorar su mantenibilidad;
- Reemplazar el sistema con un nuevo sistema.

La estrategia elegida debe depender de la calidad del sistema y su valor comercial.

Clases de sistemas heredados

- Baja calidad, bajo valor comercial.
  - Estos sistemas deben ser desechados.
- Baja calidad, alto valor comercial
  - Estos hacen una importante contribución comercial pero son costosos de mantener. Debe ser rediseñado o reemplazado si hay un sistema adecuado disponible.
- Alta calidad, bajo valor comercial.
  - Reemplace con COTS, deseche completamente o mantenga.
- Alta calidad, alto valor comercial
  - Continuar en operación utilizando el mantenimiento normal del sistema

Métricas de mantenimiento

- Métricas de producto.
  - Estas métricas describen las características del producto que de alguna forma determinan la mantenibilidad, por ejemplo el tamaño, complejidad o características del diseño.
- Métricas del proceso.
  - Las métricas del proceso pueden ser utilizadas para mejorar el desarrollo y mantenibilidad del software.
  - Algunos ejemplos incluyen la eficacia de eliminar defectos durante el desarrollo, el patrón en el que aparecen los defectos durante las pruebas o el tiempo fijo de respuesta del proceso.

Métricas sobre la complejidad

- Las predicciones de mantenibilidad pueden hacerse evaluando la complejidad de los componentes del sistema.
- Los estudios han demostrado que la mayor parte del esfuerzo de mantenimiento se gasta en un número relativamente pequeño de componentes del sistema.
- La complejidad depende de
    - Complejidad de las estructuras de control;
    - Complejidad de las estructuras de datos;
    - Objeto, método (procedimiento) y tamaño del módulo.

Métricas de procesos

- Las métricas del proceso se pueden utilizar para evaluar la mantenibilidad.
    - Número de solicitudes de mantenimiento correctivo;
    - Tiempo promedio requerido para el análisis de impacto;
    - Tiempo promedio tomado para implementar una solicitud de cambio;
    - Número de solicitudes de cambio pendientes.
- Si alguno o todos estos están aumentando, esto puede indicar una disminución en la capacidad de mantenimiento.

Métricas del producto

Índice de madurez del software

Esta métrica proporciona una indicación de la estabilidad de un producto software. A medida que el IMS se aproxima a 1, el producto comienza a estabilizarse, y por lo tanto, menos esfuerzo de mantenimiento requerirá.

$$IMS = \frac{[M_T - (F_a + F_m + F_e)]}{M_T}$$

M_T número de módulos de la versión actual

F_m número de módulos de la versión actual que han sido modificados

F_a número de módulos de la versión actual que han sido añadidos

F_e número de módulos de la versión anterior que han sido borrados en la versión actual

Métricas de calidad de la documentación

$$DensidadComentarios = n° \ JavaDocs/LoC$$

LoC : número total de líneas de código

Métricas de calidad del diseño

$$I = C_e/(C_a + C_e)$$

Donde Ce número de clases del sistema que dependen de otros sistemas Ca número de clases de otros sistemas que dependen de clases del propio sistema

$$A = Num\_Clases\_Abstractas/Clases\_Totales$$

Índice de facilidad de mantenimiento (IM)

*IM= 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') -16.2 * ln (aveLOC) + +50 * sin (sqrt(2.4 * perCM))*

Donde avecV es la media del volumen según Halstead, que es proporcional al número de operadores y operandos

V(g´) es la media de la complejidad ciclomática

V(g) = e-n+2, donde e representa el número de aristas y *n* el número de nodos, esto es, el número de posibles caminos del código.

avecLOC es la media del número de líneas de código por módulo y perCM es la media del número de código comentadas

$$X = \frac{I - A}{B}$$

Donde A es el número de fallos debidos a efectos laterales detectados y corregidos
B es el número total de fallos.

Reparabilidad:

- Un sistema software es reparable si permite la corrección de sus defectos con una cantidad de trabajo limitada y razonable.
- Compromiso entre cantidad y tamaño de los componentes o piezas.

Flexibilidad:

- Un sistema software es flexible si permite cambios para que se satisfagan nuevos requerimientos, es decir, si puede evolucionar.
- Aplicación de técnicas y metodologías apropiadas.

Puntos Clave- RESUMEN

- Hay 3 tipos de mantenimiento de software, a saber, la corrección de errores, la modificación del software para que funcione en un nuevo entorno y la implementación de requisitos nuevos o modificados.
- La reingeniería de software se ocupa de la reestructuración y la re-documentación del software para que sea más fácil de entender y cambiar.

- Refactorizar, hacer cambios en el programa que preserven la funcionalidad, es una forma de mantenimiento preventivo.
- El valor comercial de un sistema heredado y la calidad de la aplicación deben evaluarse para ayudar a decidir si un sistema debe reemplazarse, transformarse o mantenerse