DSI-Resumen.pdf

user_3706713

Diseño de sistemas

3° Ingeniería en Sistemas de Información

Facultad Regional Córdoba
Universidad Tecnológica Nacional

WORKFLOW DE ANÁLISIS

Introducción

Durante el análisis, se analizan los requerimientos capturados en la captura de requerimientos (workflow de requerimientos), refinándolos y estructurándolos. Esto se hace con el objetivo de conseguir una comprensión de los requerimientos de una forma más precisa y una descripción de los mismos que sea fácil de mantener y que nos ayude a estructurar el sistema entero.

En el workflow de requerimientos, es probable que aún queden aspectos sin resolver relativos a los requerimientos del sistema. Este es el precio que hay que pagar por el uso del lenguaje intuitivo pero impreciso del cliente durante la captura de requerimientos. El propósito fundamental del análisis es resolverlos analizando los requerimientos con mayor profundidad, pero con la gran diferencia de que puede utilizarse el lenguaje del desarrollador para describir los resultados.

MODELO DE CASOS DE USO	MODELO DE ANÁLISIS
• Descrito en lenguaje del cliente. • Vista externa del sistema. • Estructurado por casos de uso • Utilizado como contrato entre el cliente y los desarrolladores sobre que debería y que no debería hacer el sistema. • Puede contener redundancias, inconsistencias, etc. entre requerimientos. • Captura la funcionalidad del sistema, incluida la funcionalidad significativa para la arquitectura. • Define casos de uso que se analizarán con más profundidad en el modelo de análisis.	• Descrito en lenguaje del desarrollador. • Vista interna del sistema. • Estructurado por clases y paquetes estereotipados. • Utilizado por los desarrolladores para comprender como debería darse forma al sistema, es decir, como debería ser diseñado e implementado. • No debería contener redundancias, inconsistencias, etc. entre requerimientos. • Esboza como llevar a cabo la funcionalidad dentro del sistema, incluida la funcionalidad significativa para la arquitectura; sirve como una primera aproximación al diseño. • Define realizaciones de casos de uso, y cada una de ellas representa el análisis de un caso de uso de uso del modelo de casos de uso.

El análisis

El lenguaje que utilizamos en el análisis se basa en un modelo de objetos conceptual, que llamamos **modelo de análisis**. Este modelo, nos ayuda a refinar los requerimientos y nos permite razonar sobre los aspectos internos del sistema.

El modelo de análisis, también nos ayuda a estructurar los requerimientos y nos proporciona una estructura centrada en el mantenimiento, en aspectos tales como la flexibilidad ante los cambios y la reutilización. Esta estructura también sirve como entrada en las actividades de diseño e implementación, ya que se trata de preservar dicha estructura mientras le damos forma al sistema. Por esto, el modelo de análisis puede considerarse una primera aproximación al modelo de diseño, aunque es un modelo por sí mismo.

El modelo de análisis hace abstracciones y evita resolver algunos problemas y tratar algunos requerimientos que pensamos que es mejor en posponer al diseño y a la implementación. Por ello, no siempre se puede conservar la estructura proporcionada por el análisis, ya que el diseño debe considerar la plataforma de implementación: lenguaje de programación, sistemas operativos, frameworks, sistemas heredados, etc.

El análisis no es diseño ni implementación

El diseño y la implementación son mucho más que el análisis, por lo que se requiere una separación de intereses. El diseño y la implementación se preocupan en dar forma al sistema de manera que dé vida a los requerimientos (funcionales y no funcionales) que incorpora.

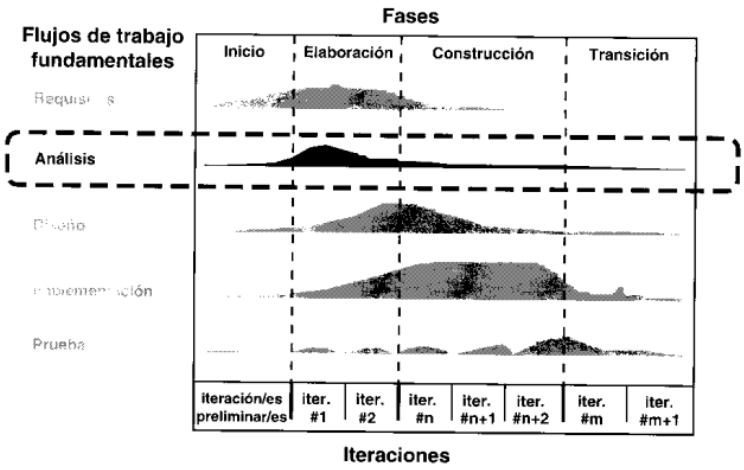
Antes de comenzar a diseñar e implementar deberíamos contar con una compresión precisa y detallada de los requerimientos. Si contamos con una estructura de requerimientos que puede utilizarse como entrada para dar forma al sistema, mejor. Esto se consigue mediante el análisis.

Cuándo hacer análisis

- Mediante la realización separada del análisis, en lugar de llevarlo a cabo como parte integrada en el diseño y la implementación, podemos analizar sin grandes costes una gran parte del sistema. Se puede utilizar el resultado para planificar el trabajo de diseño e implementación subsiguiente en varios incrementos sucesivos, donde cada incremento sola diseña e implementa una pequeña parte del sistema, o quizá en varios incrementos concurrentes.
- El análisis proporciona una visión más general del sistema que puede ser más difícil de obtener mediante el estudio de los resultados del diseño y la implementación, debido a que contienen demasiados detalles. Una visión general como esa puede ser muy valiosa para recién llegados al sistema o para desarrolladores que en general mantienen el sistema.
- Algunas partes del sistema tienen diseños y/o implementaciones alternativas. El modelo de análisis puede proporcionar una vista conceptual, precisa y unificadora de esas implementaciones alternativas.
- El sistema se construye utilizando un sistema heredado complejo. La reingeniería de este sistema heredado, o de una parte él, puede hacerse en términos del modelo de análisis de manera que los desarrolladores puedan comprender el sistema heredado sin tener que profundizar en los detalles de su diseño e implementación, y construir el nuevo sistema utilizando el heredado como un bloque de construcción reutilizable.

El papel del análisis en el ciclo de vida del software

Las iteraciones iniciales de la elaboración se centran en el análisis.



Cómo utilizar el análisis

La manera exacta de ver y emplear el análisis puede diferir de un proyecto a otro. Tres formas:

1. El proyecto utiliza el modelo de análisis para describir los resultados de análisis, y mantiene la consistencia de este modelo a lo largo de todo el ciclo de vida del software.

2. El proyecto utiliza el modelo de análisis para describir los resultados del análisis pero considera a este modelo como una herramienta intermedia o transitoria. Cuando el diseño e implantación están en marcha durante la fase de construcción, se deja de actualizar el análisis.
3. El proyecto no utiliza en absoluto el modelo de análisis para describir los resultados del análisis. En cambio, el proyecto analiza los requerimientos como parte integrada en la captura de requerimientos o en el diseño.

Al elegir entre las dos primeras variantes, hay que sopesar las ventajas de mantener el modelo de análisis con el coste de mantenerlo durante varias iteraciones y generaciones. Hay que realizar un análisis costo/beneficio correcto y decidir si se debiera dejar de actualizar el modelo de análisis o conservarlo y mantenerlo durante el resto de la vida del sistema.

La tercera variante se emplea en casos excepcionales para sistemas extraordinariamente simples.

La segunda opción suele ser la más usada, ya que el análisis se centra en la fase de elaboración.

Artefactos de análisis

1. Modelo de análisis
2. Clase del análisis
3. Realización de caso de uso-análisis
4. Paquete del análisis
5. Descripción de la arquitectura (vista del modelo de análisis)

Modelo de análisis

(ver atrás)

Clase de análisis

Representa una abstracción de una o varias clases y/o subsistemas del diseño del sistema.

- Una clase de análisis se centra en el tratamiento de los requerimientos funcionales y pospone los no funcionales.
- Son evidentes en el contexto del dominio del problema, más “conceptual”.
- Su comportamiento se define mediante responsabilidades en un alto nivel y menos formal. No se define mediante una interfaz en términos de operaciones y de sus firmas.
- Una clase de análisis define atributos en un alto nivel. Los tipos de los atributos son conceptuales y reconocibles en el dominio del problema. En el diseño los tipos pertenecen al lenguaje de programación.
- Una clase de análisis participa en relaciones, aunque esas relaciones son más conceptuales que las de diseño e implementación.
- Las clases de análisis siempre encajan en uno de los tres estereotipos básicos: de interfaz, de control o de entidad.

Estos tres estereotipos están estandarizados en UML y se utilizan para ayudar a los desarrolladores a distinguir el ámbito de las diferentes clases. Son:

- **Clases de interfaz (*boundary*):** se utilizan para modelar la interacción entre el sistema y sus actores. Esta interacción implica a menudo recibir (y presentar) información y peticiones de (y hacia) los usuarios y los sistemas externos.
  Representan a menudo abstracciones de ventanas, formularios, paneles, interfaces de comunicaciones, interfaces de impresoras, sensores, etc. Deberían asociarse con un al menos un actor y viceversa.
- **Clases de entidad:** se utilizan para modelar información que posee una vida larga y que es a menudo persistente. Modelan la información y el comportamiento asociado a algún fenómeno o concepto, como una persona, un objeto del mundo real, o un suceso del mundo real.

En la mayoría de los casos, las clases de entidad se derivan directamente de una clase de entidad del negocio (o de una clase del dominio). Una diferencia fundamental entre clases de entidad y clases de entidad de negocio es que las primeras consideran objetos manejados por el sistema. En consecuencia, las clases de entidad reflejan la información de un modo que beneficia a los desarrolladores al diseñar e implementar el sistema, incluyendo su soporte de persistencia.
Modelan información relevante para el dominio y que dura en el tiempo, siendo candidatas a ser persistentes mediante bases de datos.

- **Clases de control:** representan coordinación, secuencia, transacciones, y control de otros objetos y se usan con frecuencia para encapsular el control de un caso de uso en concreto. También se utilizan para representar derivaciones y cálculos complejos, como la lógica de negocio que no puede asociarse con ninguna clase de entidad concreta.
  Los aspectos dinámicos del sistema se modelan con clases de control, debido a que ellas manejan y coordinan las acciones y los flujos de control principales, y delegan trabajo a otros objetos.
  Son el nexo entre las clases de entidad y de interfaz, ya que éstas no se relacionan directamente.

Realización de caso de uso-análisis

Una realización de caso de uso-análisis es una colaboración dentro del modelo de análisis que describe cómo se lleva a cabo y se ejecuta un caso de uso determinado en término de las clases del análisis y de sus objetos del análisis en interacción.

Poseen una descripción textual del flujo de sucesos, diagramas de clases que muestran sus clases de análisis participantes, y diagramas de interacción que muestran la realización de un flujo o escenario particular del caso de uso en términos de interacciones de objetos del análisis. Se centra de manera natural en los requerimientos funcionales, por ende, puede posponer el tratamiento de los requerimientos no funcionales hasta el diseño y la implementación.

Paquetes del análisis

Los paquetes del análisis proporcionan un medio para organizar los artefactos del modelo de análisis en piezas manejables. Puede estar compuesto de clases de análisis, de realizaciones de casos de uso de análisis y de otros paquetes (recursivamente).

Los paquetes de análisis deberían ser cohesivos (contenidos fuertemente relacionados) y deberían ser débilmente acoplados (mínima dependencia de uno de otros).

Tienen las siguientes características:

- Los paquetes de análisis pueden representar una separación de intereses de análisis.
- Los paquetes de análisis deberían crearse basándose en los requerimientos funcionales y en el dominio del problema.
- Los paquetes del análisis probablemente se convertirán en subsistemas en las dos capas de aplicación superiores del modelo de diseño, o se distribuirán entre ellos.

Descripción de la arquitectura (vista del modelo de análisis)

La descripción de la arquitectura contiene una **vista de la arquitectura del modelo de análisis**, que muestra sus artefactos significativos para la arquitectura.

Artefactos que se consideran significativos para la arquitectura:

- La descomposición del modelo de análisis en paquetes de análisis y sus dependencias.
- Las clases fundamentales del análisis (entidad, control e interfaz).
- Realizaciones de caso de uso que describen cierta funcionalidad importante y crítica.

Trabajadores del análisis

1. Arquitecto.
2. Ingeniero de casos de uso.
3. Ingeniero de componentes.

Arquitecto

El arquitecto es el responsable de la integridad del modelo de análisis, garantizando que éste sea correcto, consistente y legible como un todo.

El modelo de análisis es correcto cuando realiza la funcionalidad descrita en el modelo de casos de uso, y sólo esa funcionalidad.

El arquitecto también es responsable de la arquitectura, es decir, de la existencia de sus partes significativas para la arquitectura tal y como se muestran en la vista de la arquitectura del modelo.

El arquitecto no es responsable del desarrollo y mantenimiento continuo de los diferentes artefactos del modelo de análisis.

Ingeniero de casos de uso

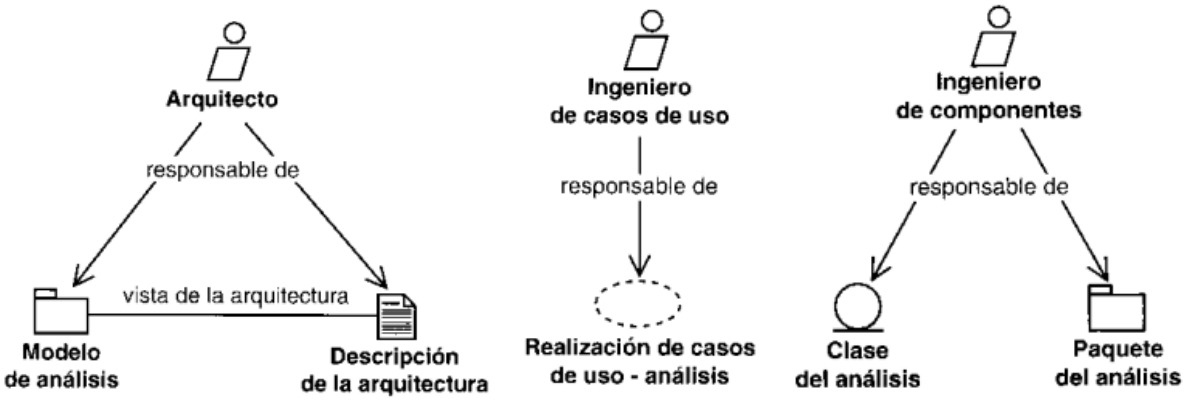
Un ingeniero de casos de uso es responsable de la integridad de una o más realizaciones de caso de uso, garantizando que cumplen los requerimientos que recaen sobre ellos.

No es responsable de las clases de análisis ni de las relaciones que se usan en la realización del caso de uso.

Ingeniero de componentes

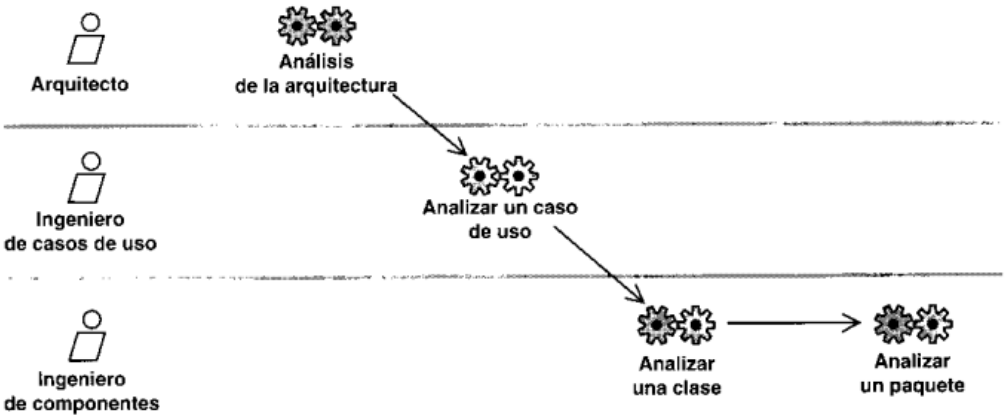
El ingeniero de componentes define y mantiene las responsabilidades, atributos, relaciones y requerimientos especiales de una o varias clases del análisis, asegurándose de que cada clase del análisis cumple con los requerimientos que se esperan de ella de acuerdo a las realizaciones de caso de uso en las que participa.

También se encarga de mantener la integridad de uno o varios paquetes del análisis.



Flujo de trabajo (actividades)

1. Los arquitectos comienzan con la creación del modelo de análisis, identificando los paquetes de análisis principales, las clases de entidad evidentes y los requerimientos comunes.
2. Los ingenieros de caso de uso realizan cada caso de uso en término de las clases de análisis participantes exponiendo los requerimientos de comportamiento de cada clase.
3. Los ingenieros de componentes especifican estos requerimientos de comportamiento y los integran dentro de cada clase creando responsabilidades, atributos y relaciones consistentes para cada clase.
4. Durante el análisis, los arquitectos identifican de manera continua nuevos paquetes de análisis, clases y requerimientos comunes a medida que el modelo evoluciona, y los ingenieros de componentes refinan y mantienen continuamente los paquetes del análisis.



Análisis de la arquitectura (por arquitecto)

El propósito del análisis de la arquitectura es esbozar el modelo de análisis y la arquitectura mediante la identificación de paquetes de análisis, clases de análisis evidentes y requerimientos especiales comunes.

Identificación de paquetes del análisis

Pueden identificarse inicialmente como una forma de dividir el trabajo de análisis, o bien encontrarse a medida que el modelo de análisis evoluciona.

Una identificación inicial de los paquetes de análisis se basa en los requerimientos funcionales y en el dominio del problema. Para ello, se asigna un número de casos de uso a un paquete y luego se realiza la funcionalidad correspondiente a ese paquete.

Estas asignaciones de casos de uso a paquetes pueden hacerse:

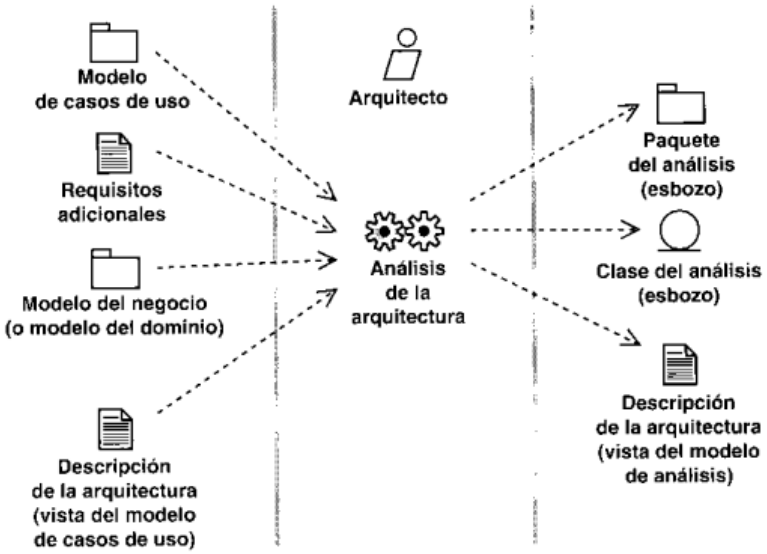
- Los casos de uso requeridos para dar soporte a un determinado proceso de negocio.
- Los casos de uso requeridos para dar soporte a un determinado actor del sistema.
- Los casos de uso que están relacionados mediante relaciones de generalización y de extensión.

Identificación de clases de entidad evidentes

Consiste en una propuesta preliminar de las clases de entidad más importantes basado en las clases del dominio o las entidades de negocio que se identificaron durante la captura de requerimientos.

Identificación de requerimientos especiales comunes

Un requerimiento especial es un requerimiento que aparece durante el análisis y que es importante anotar de forma que pueda ser tratado adecuadamente en las actividades de diseño e implementación (req. no funcionales).



Analizar un caso de uso (por ing. de casos de uso)

Analizamos un caso de uso para:

- Identificar clases de análisis cuyos objetos son necesarios para llevar a cabo el caso de uso.
- Distribuir el comportamiento del caso de uso entre los objetos del análisis que interactúan.
- Capturar requerimientos especiales sobre la realización del caso de uso.

Identificación de clases de análisis

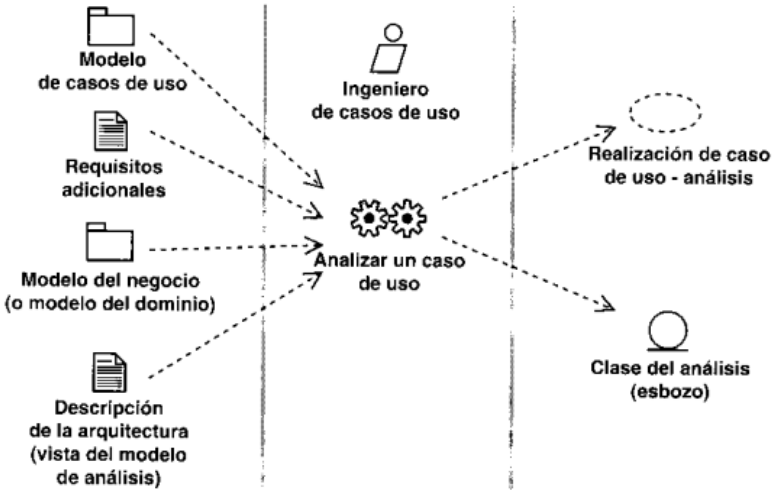
Se identifican las clases de entidad, control e interfaz necesarias para realizar los casos de uso, y se esbozan sus nombres, responsabilidades, atributos y relaciones.

Descripción de interacciones entre objetos del análisis

Cuando tenemos un esbozo de las clases necesarias para realizar un caso de uso, debemos describir cómo interactúan sus correspondientes objetos del análisis. Esto se hace mediante diagramas de colaboración que contienen las instancias de actores participantes, los objetos del análisis, y sus enlaces.

Captura de requerimientos especiales

Se capturan los requerimientos no funcionales que aparecen, pero se los posterga para el diseño e implementación.



Analizar una clase (por ing. de componentes)

Los objetivos de analizar una clase son:

- Identificar y mantener las responsabilidades de una clase de análisis, basadas en su papel en las realizaciones de caso de uso.
- Identificar y mantener los atributos y relaciones de la clase de análisis.
- Capturar requerimientos especiales sobre la realización de la clase de análisis.

Identificar responsabilidades

Las responsabilidades de una clase pueden recopilarse combinando todos los roles que cumple en diferentes realizaciones de caso de uso.

Identificar atributos

Un atributo especifica una propiedad de una clase de análisis, y normalmente es necesario para las responsabilidades de su clase.

Identificar asociaciones y agregaciones

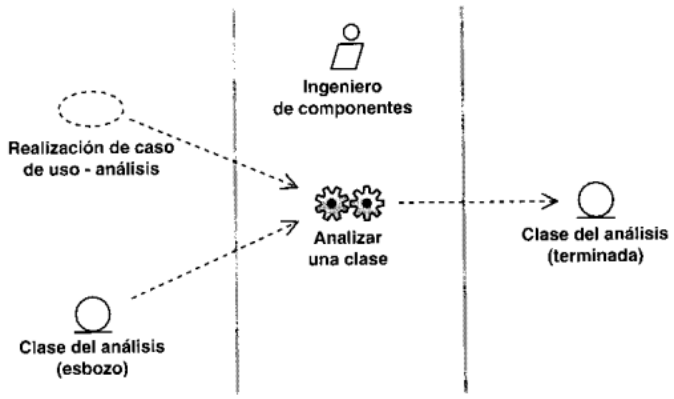
El ingeniero de componentes debería estudiar los enlaces empleados en los diagramas de colaboración para determinar que asociaciones son necesarias.

Identificar generalizaciones

Las generalizaciones deberían utilizarse en el análisis para extraer comportamiento compartido y común entre varias clases de análisis diferentes.

Captura de requerimientos especiales

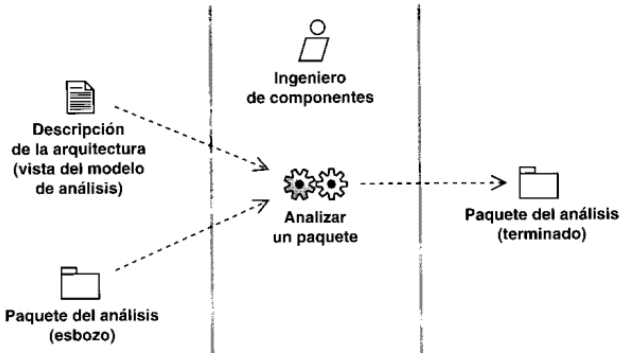
Se capturan los requerimientos no funcionales que aparecen, pero se los posterga para el diseño e implementación.



Analizar un paquete (por ing. de componentes)

Los objetivos de analizar un paquete son:

- Garantizar que el paquete de análisis es tan independiente de otros paquetes como sea posible.
- Garantizar que el paquete de análisis cumple su objetivo de realizar algunas clases de dominio o casos de uso.
- Describir las dependencias de forma que pueda estimarse el efecto de los cambios futuros.



UML

El Lenguaje Unificado de Modelado (*Unified Modeling Languaje,* UML) es un lenguaje estándar para describir planos de software. UML puede utilizarse para visualizar, especificar, construir y documentar los artefactos de un sistema que involucre una gran cantidad de software.

UML es sólo un lenguaje, y por tanto, es tan sólo una parte de un método de desarrollo de software. UML es independiente del proceso, aunque para utilizarlo óptimamente se debería usar en un proceso que fuese dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental.

UML es un lenguaje

Un lenguaje proporciona un vocabulario y las reglas para combinar palabras de ese vocabulario con el objetivo de posibilitar la comunicación. Un lenguaje de modelado es un lenguaje cuyo vocabulario y reglas se centran en la representación conceptual y física de un sistema. Un lenguaje de modelado como UML es, por tanto, un lenguaje estándar para los planos de software.

El modelado nos ayuda a la compresión del sistema. Nunca es suficiente un único modelo. Para comprender cualquier cosa, a menudo se necesitan múltiples modelos conectados entre sí.

El vocabulario y reglas de UML indican como crear y leer modelos bien formados, pero no dicen que modelos crear ni cuándo se deberían crear. Ésta es la tarea del proceso de desarrollo de software (Proceso Unificado de Desarrollo, PUD).

UML es un lenguaje para visualizar

UML es algo más que un montón de símbolos gráficos. Detrás de cada símbolo en la notación de UML hay un significado bien definido. De esta manera, un desarrollador puede escribir un modelo UML, otro desarrollador, o incluso otra herramienta, puede interpretar ese modelo sin ambigüedad.

UML es un lenguaje para especificar

Especificar se refiere a construir modelos precisos, no ambiguos y completos.

UML es un lenguaje para construir

UML no es un lenguaje de programación visual, pero sus modelos pueden conectarse de forma directa a una gran variedad de lenguajes de programación. Esto significa que es posible establecer correspondencias desde un modelo UML a un lenguaje de programación como Java, C++ o Visual Basic, o incluso a tablas e una base de datos relacional.

Esta correspondencia permite ingeniería directa e inversa:

- **Ingeniería directa:** la generación de código a partir de un modelo UML en un lenguaje de programación.
- **Ingeniería inversa:** reconstruir un modelo UML a partir de código.

UML es un lenguaje para documentar

- UML cubre la documentación de la arquitectura de un sistema y todos sus detalles.
- Proporciona un lenguaje para expresar requerimientos y pruebas.
- Proporciona un lenguaje para modelar las actividades de planificación de proyectos y gestión de versiones.

Bloques básicos de construcción

1. ELEMENTOS
   1.1. Estructurales
   1.2. De comportamiento
   1.3. De agrupación
   1.4. De anotación
2. RELACIONES
   2.1. Dependencia
   2.2. Asociación
      2.2.1.Agregación
   2.3. Generalización
   2.4. Realización
3. DIAGRAMAS

Elementos

Elementos estructurales: son las partes estáticas de un modelo, y representan conceptos o cosas materiales. Ejemplo:

1. Clase: descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica.
2. Interfaz: colección de operaciones que especifican un servicio de una clase o componente. Define un conjunto de operaciones, pero nunca la implementación de tales.
3. Colaboración: define una interacción y es una sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento mayor que la suma de los comportamientos de sus elementos.
4. Caso de uso: descripción de un conjunto de secuencias de acciones que ejecuta un sistema y que produce un resultado observable de interés para un actor particular.
5. Clase activa: es una clase cuyos objetos tienen uno o más procesos o hilos de ejecución. Clase con comportamiento concurrente.
6. Componente: es una parte modular del diseño del sistema que oculta su implementación tras un conjunto de interfaces externas.
7. Artefacto: es una parte física y reemplazable de un sistema que contiene información física (“bits”, ej. código fuente).
8. Nodo: elemento físico que existe en tiempo de ejecución y representa un recurso computacional, que por lo general tiene algo de memoria y capacidad de procesamiento.

Elementos de comportamiento: son las partes dinámicas de los modelos de UML. Ejemplo:

1. Interacción: es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular, para alcanzar un objetivo específico.
2. Máquina de estados: es un comportamiento que especifica las secuencias de estados por las que pasa un objeto o una interacción durante su vida en respuesta a eventos, junto con sus reacciones a estos eventos.
3. Actividad: es un comportamiento que especifica la secuencia de pasos que ejecuta un proceso computacional.

Elementos de agrupación: son las partes organizativas de los modelos UML. Ejemplo:

1. Paquete: es un mecanismo de propósito general para organizar el propio diseño. Es el elemento de agrupación básico para organizar un modelo UML.

Elementos de anotación: son las partes explicativas de los modelos UML. Son comentarios que se pueden aplicar para describir, clasificar y hacer observaciones sobre cualquier elemento de un modelo. Ejemplo:

1. Nota: es un símbolo para mostrar restricciones y comentarios junto a un elemento o una colección de elementos.

Relaciones

Relación de dependencia: es una relación semántica entre dos elementos, en la cual un cambio a un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (elemento dependiente).

Relación de asociación: es una relación estructural entre clases que describe un conjunto de enlaces, los cuales son conexiones entre objetos que son instancias de clases.

1. Agregación: es un tipo especial de asociación, que representa una relación estructural entre un todo y sus partes.

Relación de generalización: es una relación de especialización/generalización en la cual el elemento especializado (hijo) se basa en la especificación del elemento generalizado (padre). El hijo comparte la estructura y comportamiento del padre.

Relación de realización: es una relación semántica entre clasificadores, en donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá.

Diagramas

***Diagrama*:** es la representación gráfica de un conjunto de elementos, visualizado la mayoría de las veces como un grafo conexo de nodos (elementos) y arcos (relaciones). Los diagramas se dibujan para visualizar un sistema desde diferentes perspectivas, de forma que un diagrama es una proyección de un sistema. Un diagrama puede contener cualquier combinación de elementos y relaciones. Sin embargo, en la práctica siempre surgen un número de combinaciones habituales, las cuales son consistentes con las cinco vistas más útiles que comprenden la arquitectura de un sistema. UML incluye algunos tipos de diagramas:

1. Diagrama de clases: muestra un conjunto de clases, interfaces y colaboración, así como sus relaciones. Abarcan la vista de diseño estática de un sistema.
2. Diagrama de objetos: muestra un conjunto de objetos y sus relaciones. Representan instantáneas estáticas de instancias de los elementos de los diagramas de clases (clases). Cubren la vista de diseño estática.
3. Diagrama de componentes: representa la encapsulación de una clase, junto con sus interfaces, puertos y estructura interna, la cual está formada por otros componentes anidados y conectores. Cubren la vista de implementación estática del diseño de un sistema.
4. Diagrama de casos de uso: muestra un conjunto de casos de uso y actores y sus relaciones. Organizan y modelan el comportamiento del sistema. Cubren la vista de casos de uso estática de un sistema.
5. Diagrama de interacción: muestra una interacción, que consta de un conjunto de objetos o roles, incluyendo los mensajes que pueden ser enviados entre ellos. Cubren la vista dinámica de un sistema.
    a. *Diagrama de secuencia*: es un diagrama de interacción que resalta la ordenación temporal de los mensajes.
    b. *Diagrama de comunicación*: es un diagrama de interacción que resalta la organización estructural de los objetos o roles que envían y reciben mensajes.
6. Diagrama de estados: muestra una máquina de estados, que consta de estados, transiciones, eventos y actividades. Muestra la vista dinámica de un objeto.
7. Diagrama de actividades: muestra la estructura de un proceso u otra computación como el flujo de control y datos paso a paso. Cubren la vista dinámica del sistema.
8. Diagrama de despliegue: muestra la configuración de nodos de procesamiento en tiempo de ejecución y los artefactos que residen en ellos. Abordan la vista de despliegue estática de una arquitectura.
9. Diagrama de artefactos: muestra los constituyentes físicos de un sistema en el computador (archivos, bases de datos, etc.).

10. Diagrama de paquetes: muestra la descomposición del propio modelo en unidades organizativas y sus dependencias.
11. Diagrama de tiempos: es un diagrama de interacción que muestra los tiempos reales entre diferentes objetos o roles, en oposición a la simple secuencia relativa de mensajes.

Reglas de UML

Los bloques de construcción de UML no pueden combinarse de cualquier manera. UML tiene un número de reglas que especifican a qué debe parecerse un modelo bien formado. Un modelo bien formado es aquel que es semánticamente autoconsistente y está en armonía con todos sus modelos relacionados.

UML tiene reglas sintácticas y semánticas para:

- Nombres: como llamar a los elementos, relaciones y diagramas.
- Alcance: el contexto que da un significado específico a un nombre.
- Visibilidad: cómo se pueden ver y utilizar esos nombres por otros.
- Integridad: cómo se relacionan apropiada y consistentemente unos elementos con otros.
- Ejecución: que significa ejecutar o simular un modelo dinámico.

Es habitual que el equipo de desarrollo no sólo construya modelos bien formados, sino también modelos que sean:

- Abreviados: ciertos elementos se ocultan para simplificar la vista.
- Incompletos: pueden estar ausentes ciertos elementos.
- Inconsistentes: no se garantiza la integridad del modelo.

Mecanismos comunes de UML

1. Especificaciones.
2. Adornos.
3. Divisiones comunes.
4. Mecanismos de extensibilidad.

Especificaciones

UML es algo más que un lenguaje gráfico. Detrás de cada elemento de su notación gráfica hay una especificación que proporciona una explicación textual de la sintaxis y semántica de ese bloque de construcción.

Adornos

La mayoría de elementos UML tienen una notación gráfica única y clara que proporciona una representación visual de los aspectos más importantes del elemento.

Todos los elementos en la notación UML parten de un símbolo básico, al cual pueden añadirse una variedad de adornos específicos de ese símbolo.

Por ejemplo, a los atributos y métodos de las clases suelen ir acompañados de un símbolo que indica su visibilidad, como + (public), - (private) y # (protected).

Divisiones comunes

Al moldear sistemas orientados a objetos, el mundo a menudo se divide de varias formas.

- **División entre clase y objeto**: una clase es una abstracción; un objeto es una manifestación concreta de esa abstracción.
- **División entre interfaz e implementación**: una interfaz declara un contrato, y una implementación representa una realización de ese contrato.

- **División entre tipo y rol**: el tipo declara la clase de una entidad, como un objeto, un atributo o un parámetro. Un rol describe el significado de una entidad en un contexto, como una clase, un componente o una colaboración.

Mecanismos de extensibilidad

1. Estereotipos.
2. Valores etiquetados.
3. Restricciones.

- **Estereotipo:** extiende el vocabulario de UML, permitiendo crear nuevos bloques de construcción que derivan de los existentes pero que son específicos a un problema.
- **Valor etiquetado:** extiende las propiedades de un estereotipo de UML, permitiendo añadir nueva información en la especificación de ese estereotipo.
- **Restricción:** extiende la semántica de un bloque de construcción de UML, permitiendo añadir nuevas reglas o modificar las existentes.

En conjunto, estos tres mecanismos de extensibilidad permiten configurar y extender UML para las necesidades de un proyecto. Permiten a UML adaptarse a nuevas tecnologías de software.

Arquitectura

Arquitectura: *conjunto de decisiones significativas que tomamos sobre el producto para cumplir con los requerimientos. Explica cómo se satisfacen los requerimientos funcionales y no funcionales.*

Diferentes usuarios siguen diferentes agendas en relación con el proyecto, y cada uno mira al sistema de formas diferentes en diversos momentos a lo largo de la vida del proyecto. La arquitectura de un sistema es el artefacto más importante que puede emplearse para manejar estos diferentes puntos de vista y controlar el desarrollo iterativo e incremental de un sistema.

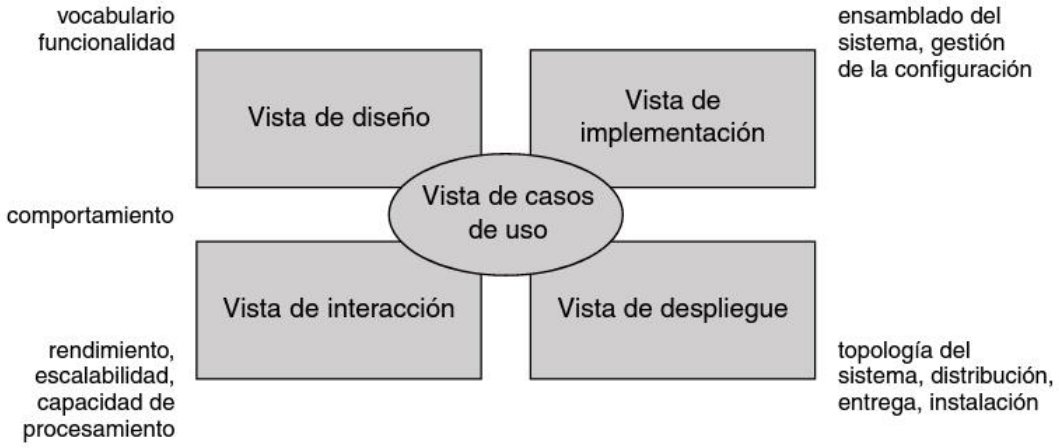
La arquitectura no tiene que ver solamente con la estructura y el comportamiento, sino también con el uso, la funcionalidad, el rendimiento, la capacidad de adaptación, las restricciones económicas y tecnológicas, etc.

Vistas

La arquitectura de un sistema puede describirse mejor a través de cinco vistas interrelacionadas. Cada vista es una proyección de la organización y la estructura del sistema, centrada en un aspecto particular del mismo.

- **Vista de casos de uso*:*** comprende los casos de uso que describen el comportamiento del sistema tal y como es percibido por los usuarios finales, analistas y encargados de las pruebas. Con UML, los aspectos estáticos de ésta vista se capturan en los diagramas de casos de uso; los aspectos dinámicos de esta vista se capturan en los diagramas de interacción, diagramas de estados y diagramas de actividades. *Comportamiento.*
- **Vista de diseño*:*** comprende las clases, interfaces y colaboraciones que forman el vocabulario del problema y su solución. Soporta principalmente los requerimientos funcionales del sistema. Con UML, los aspectos estáticos se capturan con diagramas de clases y de objetos; los dinámicos ídem a la vista de casos de uso. *Vocabulario.*
- **Vista de interacción (o de procesos):** muestra el flujo de control entre las diversas partes del sistema, incluyendo mecanismos de concurrencia y sincronización. Con UML, los aspectos estáticos y dinámicos se capturan con los mismos diagramas que en la vista de diseño, pero haciendo énfasis en clases activas que controlan el sistema y los mensajes que fluyen entre ellas. *Flujo de control.*
- **Vista de implementación*:*** comprende los artefactos que se utilizan ensamblar y poner en ejecución el sistema físico. Con UML, los aspectos estáticos de esta vista se capturan con el diagrama de artefactos; los dinámicos ídem vista de casos de uso. *Ensamblado.*

- **Vista de despliegue:** contiene los nodos que forman la topología de hardware sobre la que se ejecuta el sistema. Con UML, los aspectos estáticos se capturan con el diagrama de despliegue; los dinámicos ídem a la vista de casos de uso. *Topología de hardware.*



Cada una de estas vistas puede existir por sí misma, de forma que diferentes usuarios puedan centrarse en las cuestiones de la arquitectura del sistema que más les interesen. Además, estas cinco vistas interactúan entre sí.

Ciclo de vida del desarrollo de software

Proceso de desarrollo: *conjunto de actividades que transforma los requerimientos de un cliente en un sistema de software. Es un entorno genérico que se puede especificar según el producto, el ciclo de vida, etc.*

Entorno de proceso genérico: *que sirve para cualquier software en cualquier situación.*

UML es independiente del proceso, lo que significa que no está ligado a ningún ciclo de vida de desarrollo de software particular. Sin embargo, para obtener el máximo beneficio de UML, se debería considerar un proceso que fuese: dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental.

- **Dirigido por casos de uso:** significa que los casos de uso se utilizan como un artefacto básico para establecer el comportamiento deseado del sistema, para verificar y validar la arquitectura del sistema, para las pruebas y para la comunicación entre las personas involucradas en el proyecto. Se dice que los casos de uso guían el proceso de desarrollo.
- **Centrado en la arquitectura:** significa que la arquitectura del sistema se utiliza como un artefacto básico para conceptualizar, construir, gestionar y hacer evolucionar el sistema en desarrollo.
- **Iterativo e incremental**
    - Proceso iterativo: es aquel que involucra la gestión de un flujo de versiones ejecutables.
    - Proceso incremental: mediante la arquitectura se integran las versiones, donde cada nuevo ejecutable incorpora mejoras incrementales sobre los otros.

Este proceso dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental puede descomponerse en fases. Una **fase** es el intervalo de tiempo entre dos hitos importantes del proceso.

Fases

Hay cuatro fases en el ciclo de vida del desarrollo de software: inicio, elaboración, construcción y transición.

1. **Inicio (o concepción):** se desarrolla una descripción final del producto a partir de una buena idea y se presenta el análisis de negocio para el producto.
2. **Elaboración:** se especifican en detalle la mayoría de los casos de uso del producto (requerimientos) y se diseña la arquitectura del sistema.
3. **Construcción:** se crea el producto.

4. **Transición*:*** el producto se convierte en versión beta. Un número reducido de usuarios con experiencia prueba el producto e informa los defectos y deficiencias. Los desarrolladores corrigen los problemas e incorporan algunas mejoras.

Un elemento que distingue a este proceso y que afecta a las cuatro fases es una **iteración**.

Iteración: es un conjunto bien definido de actividades, con un plan y unos criterios de evaluación bien establecidos, que acaba en un sistema que puede ejecutarse, probarse y evaluarse.

Diagramas

Diagrama: es una representación gráfica de un conjunto de elementos, que la mayoría de las veces se dibuja como un grafo conexo de nodos (elementos) y arcos (relaciones). Los diagramas se utilizan para visualizar un sistema desde diferentes perspectivas.

Los diagramas de UML se dividen en dos categorías, diagramas estructurales y diagramas de comportamiento.

- **Diagramas estructurales*:*** sirven para representar las partes estáticas de un sistema. Ejemplos: de clase, de componentes, de estructura compuesta, de objetos, de despliegue y de artefactos.
- **Diagramas de comportamiento*:*** sirven para representar las partes dinámicas de un sistema. Ejemplo: de casos de uso, de secuencia, de comunicación, de estados y de actividades.

(Ver diagramas en bloques básicos de construcción)

Para cualquier sistema, los diagramas se organizan en paquetes, excepto en los triviales.

Interacciones

Interacción: es un comportamiento que incluye un conjunto de mensajes que se intercambian un conjunto de objetos dentro de un contexto para lograr un propósito.

Las interacciones se utilizan para modelar los aspectos dinámicos de las colaboraciones, que representan sociedades de objetos que desempeñan roles específicos, y colaboran entre sí para desarrollar un comportamiento mayor que la suma de sus elementos.

Cada interacción puede modelarse de dos formas: destacando la ordenación temporal de los mensajes, o destacando la secuencia de mensajes en el contexto de una organización estructural de objetos.

En UML, los aspectos dinámicos de un sistema se modelan mediante interacciones. Una interacción establece el escenario para su comportamiento presentando todos los objetos que colaboran para realizar alguna acción. Las interacciones incluyen mensajes enviados entre objetos. Un mensaje implica la invocación de una operación o el envío de una señal; un mensaje también puede incluir la creación o destrucción de otros objetos.

Contexto

Una interacción puede aparecer siempre que haya unos objetos enlazados a otros. Las interacciones aparecerán en la colaboración de objetos existentes en el contexto de un sistema o subsistema, en el contexto de una operación y en el contexto de una clase.

Objetos y roles

Los objetos que participan en una interacción son o bien elementos concretos (algo del mundo real) o bien elementos prototípicos (instancias).

En el contexto de una interacción podemos encontrar instancias de clases, componentes, nodos y casos de uso. Aunque las clases abstractas y las interfaces, por definición, no pueden tener instancias directas, podemos representar instancias de estos elementos en una interacción (clases hijas concretas o clases que realicen la interfaz).

Enlaces y conectores

Enlace: conexión semántica entre objetos. Es una instancia de una asociación.

Un enlace especifica un camino a lo largo del cual un objeto puede enviar un mensaje a otro objeto (o a sí mismo).

Mensajes

Mensaje: es la especificación de una comunicación entre objetos que transmite información, con la expectativa de que se desencadenará una actividad.

Cuando se pasa un mensaje, su recepción produce normalmente una acción. Una acción puede producir un cambio en el estado del destinatario y en los objetos accesibles desde él.

Se pueden modelar varios tipos de acciones:

- Llamada: invoca una operación sobre un objeto.
- Retorno: devuelve un valor al invocador.
- Envío: envía una señal a un objeto.
- Creación: crea un objeto.
- Destrucción: destruye un objeto.

El tipo más común de mensaje que se modelará será la llamada, en la cual un objeto invoca una operación de otro objeto (o de él mismo).

Los mensajes también pueden corresponderse con el envío de señales. Una señal es un valor objeto que se comunica de manera asíncrona a un objeto destinatario. Después de enviar la señal, el objeto que la envió continúa su propia ejecución. Cuando el objeto destinatario recibe la señal, él decide qué hacer con ella. Normalmente las señales disparan transiciones en la máquina de estados del receptor.

Secuenciación

Cuando un objeto envía un mensaje a otro, el objeto receptor puede, a su vez, enviar un mensaje a otro objeto, el cual puede enviar un mensaje a otro objeto diferente, y así sucesivamente. Este flujo de mensajes forma una secuencia. El inicio de cada secuencia tiene su origen en algún proceso o hilo que la contiene.

Los mensajes se ordenan en secuencia conforme sucedan en el tiempo. Para visualizar mejor la secuencia de mensajes, se puede expresar explícitamente la posición de un mensaje con relación al inicio de la secuencia, precediéndolo de un número de secuencia, separado por dos puntos.

Creación, modificación y destrucción

La mayoría de las veces, los objetos que participan en una interacción existen durante todo el tiempo que dura la interacción. Sin embargo, algunas interacciones pueden conllevar la creación y destrucción de objetos. Esto también se aplica a los enlaces.

En un diagrama de secuencia se representan de manera explícita la vida, la creación y destrucción de objetos o roles a través de la longitud vertical de sus líneas de vida. Dentro de un diagrama de comunicación, la creación y la destrucción deben indicarse con notas.

Representación

Los roles y los mensajes implicados en una interacción se pueden visualizar de dos formas: destacando la ordenación temporal de sus mensajes, o destacando la organización estructural de los roles que envían y reciben mensajes. El primero se llama diagrama de secuencia, y el segundo, diagrama de comunicación.

Los diagramas de secuencia y de comunicación son similares, lo que significa que podemos partir de uno de ellos y transformarlo en el otro, aunque a veces muestren información diferente. Los diagramas de secuencia permiten

modelar la línea de vida de un objeto, mientras que los de comunicación permiten modelar los enlaces estructurales que pueden existir entre los objetos de la interacción.

- **Diagrama de secuencia*:*** destaca la ordenación temporal de los mensajes. Un mensaje se representa con una flecha que va de una línea de vida hasta la otra. Tienen dos características que los distinguen de los de comunicación:
  - Línea de vida: es una línea discontinua vertical que representa la existencia de un objeto a lo largo de un período de tiempo.
  - Foco de control: es un rectángulo delgado y estrecho que representa el período de tiempo durante el cual un objeto ejecuta una acción.
- **Diagrama de comunicación*:*** destaca la organización de los objetos que participan en la interacción. Tienen dos características que los distinguen de los de secuencia:
  - El camino: es una línea que se dibuja haciéndola corresponder con una asociación.
  - Número de secuencia: número que indica el orden del mensaje.

Máquinas de estados

Máquina de estados: es un comportamiento que especifica la secuencia de estados por las que pasa un objeto durante su vida, en respuesta a eventos, junto con sus respuestas a esos eventos.

Se utilizan para modelar los aspectos dinámicos de un sistema. La mayoría de las veces, esto implica modelar la vida de las instancias de una clase, un caso de uso o un sistema completo. Estas instancias pueden responder a eventos tales como señales, operaciones o el paso del tiempo. Al ocurrir un evento, tendrá lugar un cierto efecto, según el estado actual del objeto. Un **efecto** es la especificación de la ejecución de un comportamiento dentro de una máquina de estados. Conllevan la ejecución de acciones que cambian el estado de un objeto o devuelven valores. El estado de un objeto es un período de tiempo durante el cual satisface alguna condición, realiza alguna actividad o espera algún evento.

La dinámica de la ejecución se puede ver de dos formas: destacando el flujo de control entre actividades (con diagramas de actividades), o destacando los estados potenciales de los objetos y las transiciones entre esos estados (con diagramas de estados).

Mientras que en una interacción se modela una sociedad de objetos que colaboran para llevar a cabo alguna acción, una máquina de estados modela la vida de un único objeto, bien sea una instancia de una clase, un caso de uso o un sistema completo. Durante su vida, un objeto puede estar expuesto a varios tipos de eventos, como una señal, la invocación de una operación, la creación o destrucción del objeto, el paso del tiempo o el cambio en alguna condición.

Como respuesta a estos eventos, el objeto reacciona con alguna acción, y después cambia su estado a un nuevo valor.

Conceptos

Estado: es una condición o situación en la vida de un objeto durante el cual satisface alguna condición, realiza alguna actividad o espera algún evento.

Evento: es la especificación de un acontecimiento significativo situado en el tiempo y en el espacio. Es la aparición de un estímulo que puede disparar una transición de estados (o no).

Transición: es una relación entre dos estados que indica que un objeto que esté en el primer estado realizará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especificado y se satisfagan ciertas condiciones especificadas.

Actividad: es una ejecución no atómica en curso, dentro de una máquina de estados.

Acción: es una computación atómica ejecutable que produce un cambio en el estado del modelo o devuelve un valor.

Contexto

Es la mejor forma de especificar el comportamiento de los objetos que deben responder a estímulos asíncronos o cuyo comportamiento actual depende de su pasado.

Estados

Un objeto permanece en un estado durante una cantidad de tiempo finita.

Un estado tiene varias partes:

- Nombre: cadena de texto que distingue al estado de otros.
- Efectos de entrada/salida: acciones ejecutadas al entrar y salir del estado.
- Transiciones internas: transiciones que se manejan sin causar un cambio de estado.
- Subestados: estructura anidada de un estado, que engloba subestados disjuntos o concurrentes. Estados anidados dentro de otro.
- Eventos diferidos: lista de eventos que no se manejan en este estado sino que se posponen y se añaden a una cola para ser manejado por el objeto en otro estado.

En la máquina de estados de un objeto se pueden definir dos estados especiales:

- **Estado inicial*:*** indica el punto de comienzo por defecto para la máquina de estados. Se representa mediante un círculo negro.
- **Estado final*:*** indica que la ejecución de la máquina de estados o del estado que lo contiene ha finalizado. Se representa con un círculo negro dentro de un círculo blanco.

Transiciones

Una transición tiene cinco partes:

- Estado origen: el estado afectado por la transición.
- Evento de disparo: evento que provoca la transición.
- Condición de guarda: expresión booleana que se evalúa cuando la transición se activa por la recepción del evento de disparo.
- Efecto: un comportamiento ejecutable, como una acción.
- Estado destino: el estado activo tras completarse la transición.

Una transición se representa con una línea continua dirigida desde el estado origen hacia el destino. Una autotransición es una transición donde el estado origen y el estado destino son el mismo.

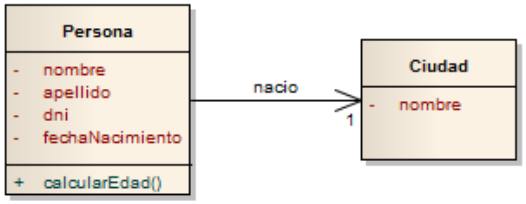
NOTA: a los diagramas de interacción se le agregan los diagramas de timing y los diagramas de descripción de interacción.

El diagrama de una máquina de estados es a nivel de clase, mientras que el de timing es a nivel de instancia (o sea a nivel de objeto).

Responsabilidades y métodos

UML define una **responsabilidad** como "un contrato u obligación de un clasificador". Las responsabilidades están relacionadas con las obligaciones de un objeto en cuanto a su comportamiento. Estas responsabilidades son de dos tipos: conocer y hacer.

- Responsabilidades del **conocer**:
  - Conocer datos privados encapsulados. *Conocer datos propios como nombre, apellido, dni, fechaNacimiento.*
  - Conocer objetos relacionados. *Conocer atributos de las clases relacionadas como nacio.*



  - Conocer las cosas que puede derivar o calcular. *Conocer datos que se pueden calcular de datos propios como la edad mediante calcularEdad() que utilizará fechaNacimiento para realizar el cálculo.*
- Responsabilidades del **hacer**:
  - Hacer algo él mismo, como crear un objeto o hacer un cálculo, y responder peticiones.
  - Iniciar una acción en otros objetos (delegar responsabilidades).
  - Controlar y coordinar actividades en otros objetos.

Una responsabilidad no es lo mismo que un método, pero los métodos se implementan para llevar a cabo responsabilidades. Las responsabilidades se implementan utilizando métodos que o actúan solos o colaboran con otros métodos u objetos.

Patrones

Un **patrón** es una descripción de un problema y la solución, a la que se da un nombre, y que se puede aplicar a nuevos contextos. Proporciona consejos sobre el modo de aplicarlo en varias circunstancias, y considera los puntos fuertes y compromisos.

Patrón: es un par problema/solución con nombre que se puede aplicar en nuevos contextos, con consejos acerca de cómo aplicarlo en nuevas situaciones y discusiones sobre sus compromisos.

Los importante de los patrones no es expresar nuevas ideas de diseño, sino justamente lo contrario. Los patrones pretenden codificar conocimientos, estilos y principios existentes y que se han probado que son válidos.

Todos los patrones, idealmente, tienen nombres sugerentes que apoyan la identificación e incorporación de ese concepto en nuestro conocimiento y memoria, y facilita la comunicación.

Patrones de asignación de responsabilidades (GRASP)

Patrones GRASP: describen los principios fundamentales del diseño de objetos y la asignación de responsabilidades, expresados como patrones. GRASP (*General Responsibility Assignment Software Patterns*).

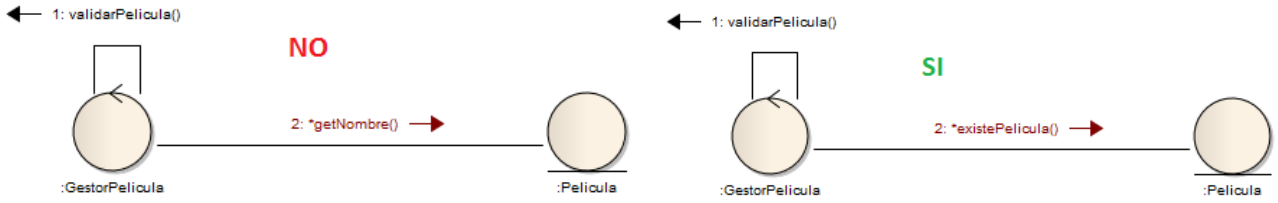
1. Experto en información.
2. Creador.
3. Alta cohesión.
4. Bajo acoplamiento.
5. Controlador.

Experto en información

Solución: asignar una responsabilidad al experto en información (la clase que tiene la información necesaria para realizar la responsabilidad).

"Poner el comportamiento lo más cerca posible de los datos", o sea, poner los métodos en la clase que tiene los datos para cumplir con el comportamiento.

Ejemplo:



En el primer ejemplo, el gestor "junta" todos los nombres de las películas y luego resuelve el problema el mismo (verificar si existe).

En el segundo ejemplo, se delega la responsabilidad a la clase Película, donde directamente el gestor le pregunta si existe, y la misma Película, utilizando su atributo nombre se encargará de resolver el problema.

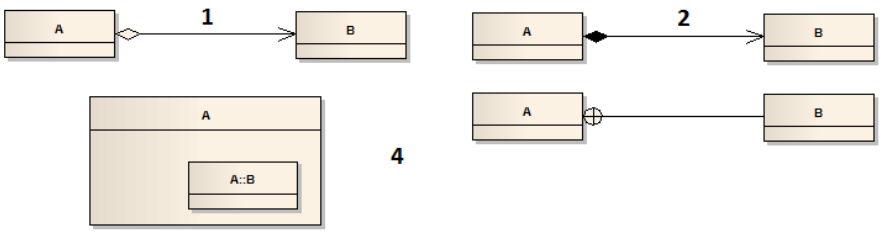
Beneficios:

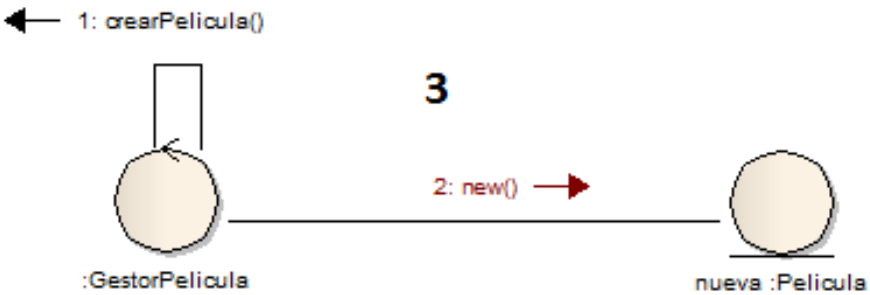
- Se mantiene el encapsulamiento de la información, puesto que los objetos utilizan su propia información para llevar a cabo las tareas. Esto conlleva a un bajo acoplamiento, lo que da lugar a sistemas más robustos y fáciles de mantener.
- Se distribuye el comportamiento entre las clases que contienen la información requerida, por tanto, se estimula las definiciones de clases más cohesivas y ligeras que son más fáciles de entender y mantener.

Creador

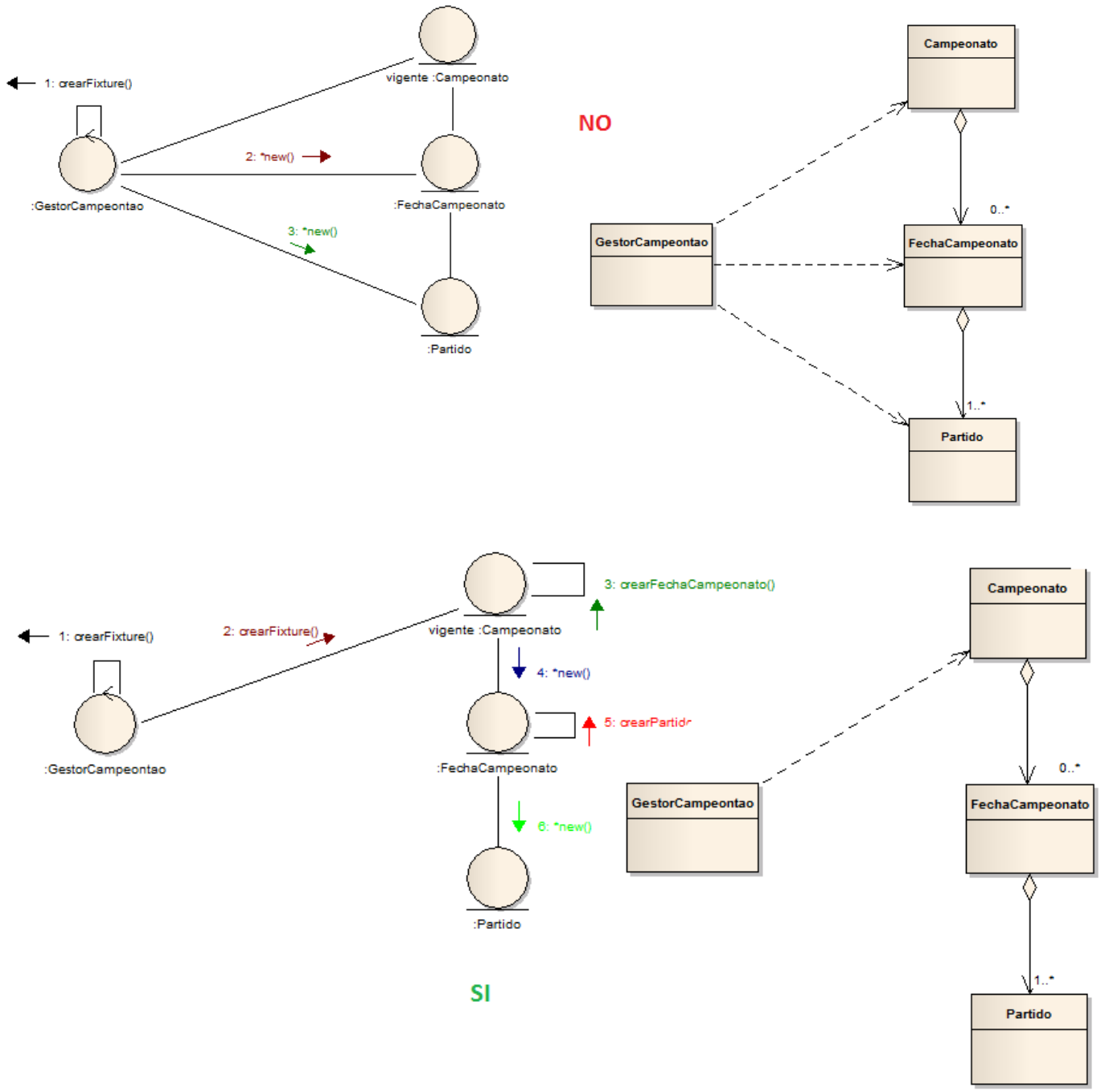
Solución: asignar a la clase A la responsabilidad de crear una instancia de la clase B si se cumple uno o más de los casos siguientes:

1. A está formado por B
2. A está compuesto por B
3. A tiene la información para crear a B
4. A contiene a B
5. A utiliza a B





Ejemplo:



Aplicando el patrón se eliminan dependencias que son innecesarias, disminuyendo el acoplamiento.

Beneficios:

- Se soporta bajo acoplamiento, lo que implica menos dependencias de mantenimiento y mayores oportunidades para reutilizar.

  No se incrementa el acoplamiento porque la clase creada es presumible que ya sea visible a la clase creadora, debido a las asociaciones existentes que motivaron su elección como creador.

Bajo acoplamiento

Solución: asignar una responsabilidad de manera que el acoplamiento permanezca bajo ☺

Acoplamiento: es una medida de la fuerza con que un elemento está conectado a, tiene conocimiento de, confía en, otros elementos. Un elemento con bajo (o débil) acoplamiento no depende de demasiados otros elementos; "demasiados" depende del contexto.

Una clase con alto (o fuerte) acoplamiento confía en muchas otras clases. Tales clases podrían no ser deseables; algunas adolecen de los siguientes problemas:

- Los cambios en las clases relacionadas fuerzan cambios locales.
- Son difíciles de entender de manera aislada.
- Son difíciles de reutilizar puesto que su uso requiere la presencia adicional de las clases de las que depende.

Es la idea de tener las clases lo menos ligadas entre sí que se pueda. De tal forma que en caso de producirse una modificación en alguna de ellas, se tenga la mínima repercusión posible en el resto de clases, potenciando la reutilización, y disminuyendo la dependencia entre las clases.

El caso extremo de Bajo Acoplamiento es cuando no existe acoplamiento entre clases. Esto no es deseable porque una metáfora central de la tecnología de objetos es un sistema de objetos conectados que se comunican mediante el paso de mensajes.

Un grado moderado de acoplamiento es normal y necesario si quiere crearse un sistema orientado a objetos, donde los objetos colaboran entre sí.

Beneficios:

- No afectan los cambios en otros componentes.
- Fácil de entender de manera aislada.
- Conveniente para reutilizar.

Alta cohesión

Solución: asignar una responsabilidad de manera que la cohesión permanezca alta.

Cohesión: es una medida de la fuerza con la que se relacionan los objetos y de grado de focalización de las responsabilidades de un elemento. Un elemento con responsabilidades altamente relacionadas, y que no hace una gran cantidad de trabajo, tiene alta cohesión.

Una clase con baja cohesión hace muchas cosas no relacionadas, o hace demasiado trabajo. Tales clases no son convenientes, adolecen de los siguientes problemas:

- Difíciles de entender.
- Difíciles de reutilizar.
- Difíciles de mantener.
- Delicadas, constantemente afectadas por los cambios.

Cada atributo y método tiene que tener sentido con el concepto de la clase. Una alta cohesión funcional provoca una mayor cantidad de clases.

Beneficios:

- Se incrementa la claridad y facilita la compresión del diseño.
- Se simplifican el mantenimiento y las mejoras.
- Se soporta a menudo bajo acoplamiento.

- Incrementa la reutilización porque una clase cohesiva se puede utilizar para un propósito muy específico.

Controlador

Solución: asignar la responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que representa una de las siguientes opciones:

- Representa el sistema global, dispositivo o subsistema.
- Representa un escenario de caso de uso en el que tiene lugar el evento del sistema.

Controlador: es un objeto que no pertenece a la interfaz de usuario, responsable de recibir o manejar un evento del sistema. Un controlador define el método para la operación del sistema.

Es un patrón que sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es la que recibe los datos del usuario y la que los envía a las distintas clases según el método llamado. Este patrón sugiere que la lógica de negocios debe estar separada de la capa de presentación, esto para aumentar la reutilización de código y a la vez tener un mayor control.

Normalmente, un controlador debería **delegar** en otros objetos el trabajo que necesita hacer, coordina o controla la actividad.

Hay dos categorías de controlador:

- ***Controlador de fachada:*** representa al sistema global, dispositivo o subsistema. Son adecuados cuando no existen "demasiados" eventos del sistema, o no es posible que la interfaz de usuario (UI) redireccione mensajes de los eventos del sistema a controladores alternativos.
- ***Controlador de casos de uso:*** hay un controlador diferente para cada caso de uso. Aquí, el controlador no es un objeto de dominio; es una construcción artificial para dar soporta al sistema (clase de fabricación pura).

Un importante corolario del patrón Controlador es que los objetos interfaz y la capa de presentación no deberían ser responsables de llevar a cabo los eventos del sistema. Las operaciones del sistema se deberían manejar en la lógica de la aplicación o capa del dominio en lugar de en la capa de interfaz del sistema. El controlador recibe la solicitud del servicio desde la capa de UI y coordina su realización, normalmente delegando a otros objetos. *Intermediario entre la interfaz y las entidades.*

Beneficios:

- Aumenta el potencial para reutilizar y las interfaces conectables: asegura que la lógica de la aplicación no se maneja en la capa de interfaz. Delegando la responsabilidad de una operación del sistema a un controlador ayuda a la reutilización de la lógica en futuras aplicaciones. Y puesto que la lógica no está ligada a la capa de interfaz, puede sustituirse por una interfaz nueva.
- Razonamiento sobre el estado de los casos de uso: a veces es necesario asegurar que las operaciones del sistema tienen lugar en una secuencia válida, o ser capaces de razonar sobre el estado actual de la actividad y operaciones del caso de uso que está en marcha.

Fabricación pura: es una creación arbitraria del diseñador, no una clase software cuto nombre se inspira en el modelo de dominio. Un controlador de caso de uso es un tipo de fabricación pura.

WORKFLOW DE DISEÑO

Introducción

En el diseño modelamos el sistema y encontramos su forma (incluida la arquitectura) para que soporte todos los requerimientos, incluyendo los requerimientos no funcionales y otras restricciones. Una entrada esencial en el diseño es el resultado del análisis, o sea el modelo de análisis, el cual proporciona una comprensión detallada de los requerimientos e impone una estructura del sistema que debemos esforzarnos por conservar los más fielmente posible al momento de darle forma al sistema.

Análisis vs Diseño

El análisis:

- Ofrece una especificación más precisa de los requerimientos.
- Se describe utilizando el lenguaje de los desarrolladores, introduciendo mayor formalismo.
- Estructura los requerimientos de modo que facilita su comprensión, su preparación, su modificación, y su mantenimiento.
- Es la primera aproximación al diseño.

El diseño:

- Es un proceso mediante el cual se aplican varias técnicas y principios con el objetivo de definir un dispositivo, un proceso o un sistema con suficiente nivel de detalle como para permitir su realización física.
- Es un proceso iterativo de transformar un modelo lógico en un modelo físico, teniendo en cuenta las restricciones del negocio.
- Es el proceso de decidir de cómo se va a construir el producto final.

MODELO DE ANÁLISIS	MODELO DE DISEÑO
• Modelo conceptual, porque es una abstracción del sistema.	• Modelo físico, porque es un plano de la implementación.
• Genérico respecto al diseño (es aplicable a varios diseños).	• No es genérico, es específico para una implementación.
• Tres estereotipos conceptuales sobre las clases: entidad, control e interfaz.	• Número indefinido de estereotipos (físicos) sobre las clases, dependiendo del lenguaje de programación.
• Menos formal.	• Más formal.
• Menos caro de desarrollar.	• Más caro de desarrollar.
• Dinámico (no muy centrado en la secuencia).	• Dinámico (muy centrado en las secuencias).
• Bosquejo del diseño del sistema, incluyendo su arquitectura.	• Manifiesto del diseño del sistema, incluyendo su arquitectura.
• Puede no estar mantenido durante todo el ciclo de vida del software.	• Debe ser mantenido durante todo el ciclo de vida del software.
• Define una entrada que es esencial para modelar el sistema.	• Da forma al sistema mientras que se intenta preservar la estructura definida en el modelo de análisis lo más posible.
• Modela la solución en términos lógicos.	• Modela la solución en términos físicos.

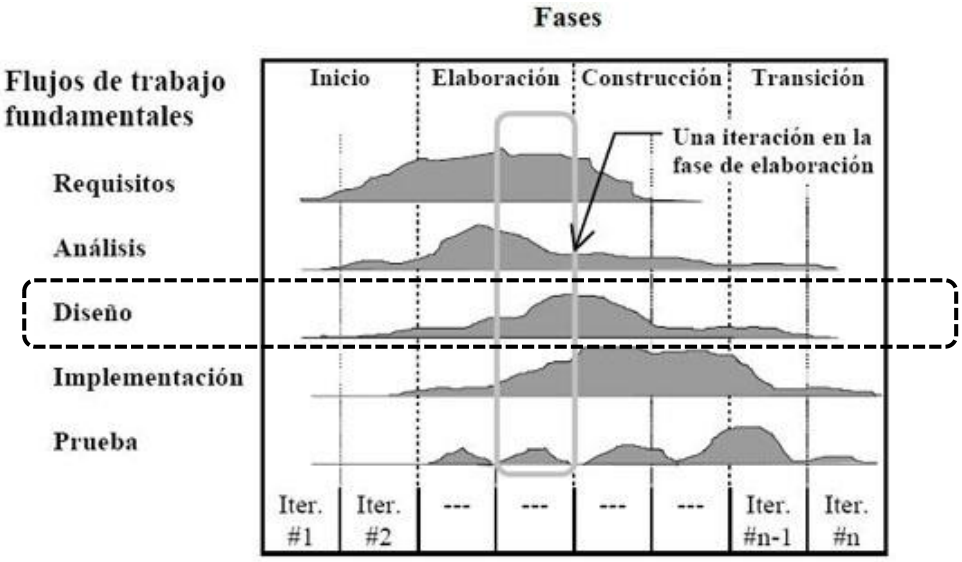
Propósitos del diseño

- Adquirir una compresión profunda de aspectos relacionados con requerimientos no funcionales y restricciones del entorno de implementación (lenguajes de programación, sistemas operativos, etc.).
- Refinar los requerimientos para subsistemas, clases e interfaces.
- Descomponer el trabajo de implementación en piezas manejables por diferentes equipos de desarrollo.
- Capturar interfaces entre subsistemas.
- Crear una abstracción de la implementación del sistema.

El papel del diseño en el ciclo de vida del software

El diseño es el centro de atención al final de la fase de elaboración y el comienzo de las iteraciones de construcción. Durante la fase de construcción, cuando la arquitectura es estable y los requerimientos están bien entendidos, el centro de atención se desplaza a la implementación.

El modelo de diseño se puede utilizar para visualizar la implementación y para soportar las técnicas de programación gráfica. A esto se lo denomina ingeniería de ida y vuelta. El diseño crea el plano para el modelo de implementación.



Artefactos de diseño

1. Modelo de diseño.
2. Clase de diseño.
3. Realización de caso de uso-diseño.
4. Subsistema de diseño.
5. Interfaz.
6. Descripción de la arquitectura (vista del modelo de diseño).
7. Modelo de despliegue.
8. Descripción de la arquitectura (vista del modelo de despliegue).

Modelo de diseño

Es un modelo de objetos que describe la realización física de los casos de uso centrándose en cómo los requerimientos funcionales y no funcionales, junto con otras restricciones de la implementación, tienen impacto en el sistema a considerar. Sirve de abstracción de la implementación del sistema.

Los casos de uso son realizados por clases de diseño y sus objetos. Esto se representa por colaboraciones en el modelo de diseño y denota realización de caso de uso-diseño.

Clase de diseño

Las clases de diseño son clases cuyas especificaciones se han completado hasta tal nivel que se pueden implementar.

- Son especificadas mediante un lenguaje de programación.
- Se especifican con frecuencia la visibilidad de los atributos y operaciones.
- Los métodos (realizaciones de operaciones) de una clase de diseño, tienen correspondencia directa con el correspondiente método en la implementación de las clases (en el código).
- Una clase de diseño puede posponer el manejo de algunos requerimientos para las subsiguientes actividades de implementación, indicándolos como requerimientos de implementación de la clase.
- Una clase de diseño puede realizar (y por tanto, proporcionar) interfaces, si tiene sentido hacerlo en el lenguaje de programación.
- Una clase de diseño puede activarse, implicando que objetos de la clase mantengan su propio hilo de control y se ejecuten concurrentemente con otros objetos activos.

Las clases de diseño provienen de dos lados:

- El ámbito del problema vía una mejora de las clases de análisis: a las clases de análisis se le añaden detalles de implementación.
- El ámbito de la solución: es el ámbito de librerías de clases de utilidad y componentes reutilizables (Time, Date, String, colecciones, middleware, frameworks, etc.).

En las clases de diseño hay que especificar: atributos (nombre, tipo, visibilidad y opcionalmente un valor predeterminado), operaciones o métodos (nombre, parámetros con tipo y tipo de retorno). Esto además sirve para generar código que después les servirá a los programadores.

Cuatro características que debe tener una clase de diseño para que se considere bien diseñada:

- Completa y suficiente: proporciona a los clientes lo que se espera de ella y contiene el conjunto esperado de operaciones y nada más.
- Sencilla: ofrece un solo servicio sencillo. A veces se "relaja" un poco por cuestiones de rendimiento.
- Alta cohesión: tiene un pequeño número de responsabilidades que están íntimamente relacionadas. Son fáciles de entender y mantener, y son reutilizables.
- Bajo acoplamiento: debería asociarse con clases si solamente tienen vínculo semántico entre ellas.

Realización de caso de uso-diseño

Una realización de caso de uso-diseño es una colaboración en el modelo de diseño que describe como se realiza un caso de uso específico, y como se ejecuta, en términos de clases de diseño y sus objetos.

Una realización de caso de uso-diseño tiene una descripción de flujo de eventos textual, diagramas de clases que muestra sus clases de diseño participantes; y diagramas de interacción que muestran la realización de un flujo o escenario concreto de un caso de uso en términos de interacción entre objetos del diseño.

Una realización de caso de uso-diseño proporciona una realización física de la realización de caso de uso-análisis para la que es trazado, y también gestiona muchos de los requerimientos no funcionales capturados en la realización de caso de uso-análisis.

Pueden posponer requerimientos hasta la siguiente actividad anotándolos como requerimientos de implementación en la realización.

- **Diagramas de clases**: contienen clases de diseño, y se utilizan conectados a una realización de caso de uso-diseño para mostrar las clases de diseño participantes, subsistemas y sus relaciones.

- **Diagramas de interacción**: muestran la interacción entre objetos para llevar a cabo el caso de uso. En el diseño es preferible utilizar diagramas de secuencia ya que es importante encontrar secuencias de interacciones detalladas y ordenadas en el tiempo.
- **Flujo de sucesos-diseño**: es una descripción textual que explica y complementa a los diagramas y a sus etiquetas.
- **Requerimientos de implementación**: requerimientos capturados en el diseño, pero que es mejor tratarlos en la implementación.

Subsistema de diseño

Son una forma de organizar los artefactos del modelo de diseño en piezas más manejables.

Un subsistema debería ser cohesivo, es decir, sus contenidos deberían estar fuertemente asociados. Además, los subsistemas deberían ser débilmente acoplados, esto es, sus dependencias entre unos y otros, o entre sus interfaces, deberían ser mínimas.

Deberían poseer las siguientes características:

- Pueden representar una separación de aspectos del diseño. Un sistema puede desarrollarse por separado por diferentes grupos de desarrolladores.
- Las dos capas de aplicación de más alto nivel y sus subsistemas dentro del modelo de diseño suelen tener trazas directas hacia paquetes y/o clases del análisis.
- Pueden representar componentes de grano grueso en la implementación del sistema.
- Pueden representar productos de software reutilizados que han sido encapsulados en ellos.
- Pueden representar sistemas heredados (o parte de ellos) encapsulándolos.

Interfaz

Las interfaces se utilizan para especificar las operaciones que proporcionaran las clases y los subsistemas de diseño.

Constituyen una forma de separar la especificación de la funcionalidad en términos de operaciones de sus implementaciones en términos de métodos.

Podemos sustituir una implementación concreta de una interfaz, como puede ser una clase o un subsistema del diseño, por otra implementación sin tener que cambiar los clientes.

Descripción de la arquitectura (vista del modelo de diseño)

La descripción de la arquitectura contiene una vista de la arquitectura del modelo de diseño que muestra los artefactos más significativos para la arquitectura, como:

- La descomposición del modelo de diseño en subsistemas, sus interfaces y las dependencias entre ellos, ya que constituyen la estructura fundamental del sistema.
- Clases del diseño fundamentales, como clases con traza con clases del análisis, clases activas, clases de diseño generales y centrarles; normalmente son clases abstractas.
- Realizaciones de caso de uso-diseño que describan alguna funcionalidad importante y crítica que debe desarrollarse pronto dentro del ciclo de vida del software, que impliquen muchas clases de diseño y por tanto tengan una cobertura amplia, posiblemente a lo largo de varios subsistemas.

Modelo de despliegue

El modelo de despliegue es un modelo de objetos que describe la distribución física del sistema en términos de cómo se distribuye la funcionalidad entre los nodos de cómputo.

- Cada nodo representa un recurso de cómputo (ej. procesador).

- Los nodos poseen relaciones que representan medios de comunicación entre ellos (ej. internet, intranet, bus, etc.).
- El modelo de despliegue puede describir diferentes configuraciones de red.
- La funcionalidad (los procesos) de un nodo se definen por los componentes que se distribuyen en ellos.
- El modelo de despliegue en sí mismo representa una correspondencia entre la arquitectura software y la arquitectura del sistema (el hardware).

Descripción de la arquitectura (vista del modelo de despliegue)

La descripción de la arquitectura contiene una vista de la arquitectura del modelo de despliegue que muestra sus artefactos relevantes para la arquitectura.

Debido a su importancia, deberían mostrarse todos los aspectos del modelo de despliegue en la vista arquitectónica, incluyendo la correspondencia de los componentes sobre los nodos tal como se identificó durante la implementación.

Trabajadores del diseño

1. Arquitecto.
2. Ingeniero de casos de uso.
3. Ingeniero de componentes.

Arquitecto

En el diseño, el arquitecto es responsable de la integridad de los modelos de diseño y de despliegue, garantizando que los modelos sean correctos, consistentes y legibles como un todo.

Los modelos son correctos cuando realizan la funcionalidad, y sólo la funcionalidad, descrita en el modelo de casos de uso, en los requerimientos adicionales y en el modelo de análisis.

También, es responsable de la arquitectura de los modelos de diseño y despliegue, es decir, de la existencia de sus partes significativas para la arquitectura.

No es responsable del desarrollo y mantenimiento continuos de los distintos artefactos del modelo de diseño.

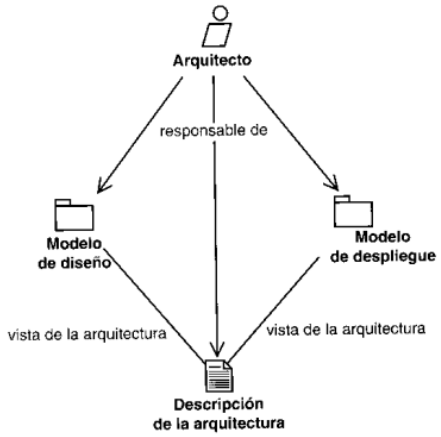
Ingeniero de casos de uso

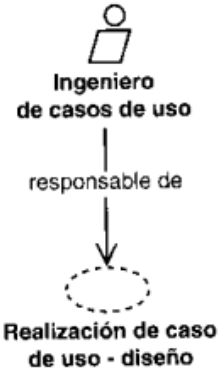
El ingeniero de casos de uso es responsable de la integridad de una o más realizaciones de caso de uso-diseño, y debe garantizar que cumplen los requerimientos que se esperan de ellos. Una realización de caso de uso-diseño debe realizar correctamente el comportamiento de su correspondiente realización de caso de uso-análisis del modelo de análisis, así como el comportamiento de su correspondiente caso de uso del modelo de casos de uso, y solo esos comportamientos.

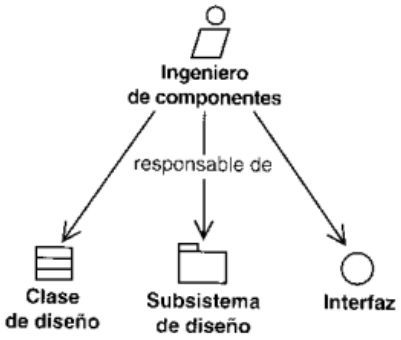
Ingeniero de componentes

El ingeniero de componentes define y mantiene las operaciones, métodos, atributos, relaciones y requerimientos de implementación de una o más clases del diseño, garantizando que cada clase cumple los requerimientos esperados de ella según las realizaciones de caso de uso en las que participa.

El ingeniero de componentes puede mantener también la integridad de uno o más subsistemas. Esto incluye garantizar que sus contenidos (clases y relaciones) son correctos, que las dependencias de otros subsistemas y/o interfaces son correctas y mínimas, y que realizan correctamente las interfaces que ofrecen.

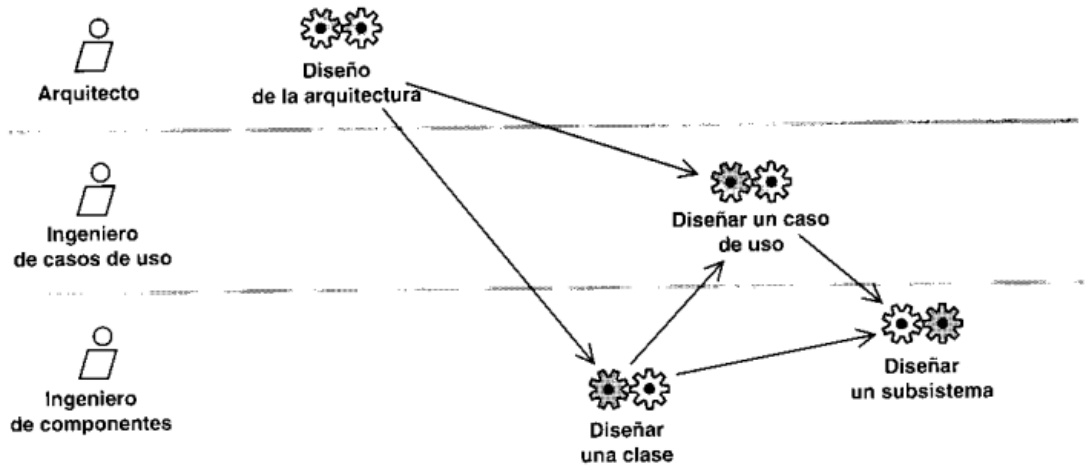






Flujo de trabajo (actividades)

1. Los arquitectos inician la creación de los modelos de diseño y de despliegue (esbozan nodos del modelo de despliegue, los subsistemas principales y sus interfaces, las clases de diseño importantes).
2. Después, los ingenieros de casos de uso realizan cada caso de uso en términos de clases y/o subsistemas del diseño participantes y sus interfaces.
3. Los ingenieros de componentes especifican a continuación los requerimientos, y los integran dentro de cada clase, bien mediante la creación de operaciones, atributos y relaciones consistentes sobre cada clase, o bien mediante la creación de operaciones consistentes en cada interfaz que proporcione el subsistema.
4. A lo largo del flujo de trabajo de diseño, los desarrolladores identificarán, a medida que evolucione el diseño, nuevos candidatos para ser subsistemas, interfaces, clases y mecanismos de diseño genérico, y los ingenieros de componentes responsables de los subsistemas individuales deberán refinarlos y mantenerlos.



Diseño de la arquitectura (por arquitecto)

El objetivo del diseño de la arquitectura es esbozar los modelos de diseño y despliegue y su arquitectura mediante la identificación de los siguientes elementos:

Identificación de nodos y configuraciones de red

Las configuraciones de red habituales utilizan un patrón de tres capas en el cual los clientes se dejan en una capa, la funcionalidad de la base de datos en otra, y la lógica de negocio en una tercera. Aspectos a tener en cuenta:

- Que nodos se necesitan. Cuál debe ser su capacidad en términos de potencia y tamaño de memoria.
- Qué tipo de conexiones debe haber entre los nodos. Que protocolos de comunicaciones se deben utilizar.

- Qué características deben tener las conexiones y los protocolos de comunicaciones, como el ancho de banda, disponibilidad, calidad.
- Si se necesita copias de seguridad de datos, capacidad de procesos redundante, etc.

Identificación de subsistemas y sus interfaces

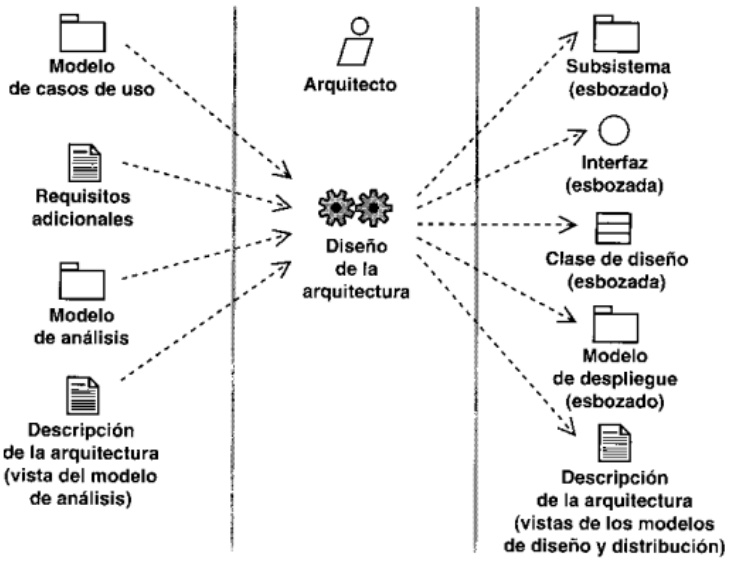
Los subsistemas pueden identificarse inicialmente como una forma de dividir el trabajo de diseño, o bien pueden irse encontrando a medida que el modelo de diseño evoluciona y va creciendo hasta convertirse en una gran estructura que debe ser descompuesta. Algunos subsistemas representan productos reutilizados y otros son recursos existentes en la empresa.

Identificación de clases de diseño significativas para la arquitectura

Los desarrolladores deberían evitar el identificar demasiadas clases en esta etapa o el quedar atrapados en demasiados detalles. Sería suficiente con un esbozo inicial de las clases significativas para la arquitectura, como las clases activas.

Identificación de mecanismos genéricos de diseño

El resultado es un conjunto de mecanismos genéricos de diseño que pueden manifestarse como clases, colaboraciones, subsistemas. Los requerimientos a tratarse suelen estar relacionados con: persistencia, distribución transparente de objetos, características de seguridad, detección y recuperación de errores, etc.



Diseño de un caso de uso (por ing. de casos de uso)

Los objetivos de diseñar un caso de uso son:

- Identificar las clases del diseño y/o subsistemas cuyas instancias son necesarias para llevar a cabo el flujo de sucesos del caso de uso.
- Distribuir el comportamiento del caso de uso entre los objetos del diseño que interactúan y/o entre los subsistemas participantes.
- Definir los requerimientos sobre las operaciones de las clases del diseño y/o sobre los subsistemas y sus interfaces.
- Capturar los requerimientos de implementación del caso de uso.

Identificar las clases del diseño participantes

Se identifican las clases de diseño necesarias para realizar el caso de uso.

- Estudiar las clases del análisis que participan en la correspondiente realización de caso de uso-análisis.

- Estudiar los requerimientos especiales de la correspondiente realización de caso de uso-análisis, identificando las clases del diseño que los cumplen.

Descripción de las interacciones entre objetos del diseño

Debemos describir cómo interactúan los objetos del diseño correspondientes a las clases del diseño. Esto se hace mediante diagramas de secuencia que contienen instancias de los actores, los objetos del diseño y las transmisiones de mensajes entre éstos, que participan en el caso de uso.

Identificación de subsistemas e interfaces participantes

A veces es más apropiado diseñar un caso de uso en término de los subsistemas y/o interfaces que participan en él:

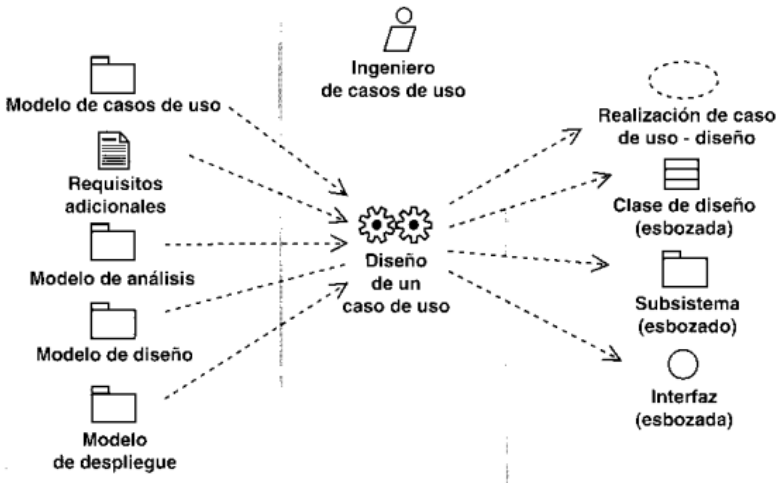
- Estudiar las clases del análisis que participen en las correspondientes realizaciones de caso de uso-análisis. Identificar los paquetes del análisis que contienen a esas clases del análisis, si existen. Después, identificar los subsistemas del diseño que poseen una traza hacia eses paquetes del análisis.
- Estudiar los requerimientos especiales de las correspondientes realizaciones de caso de uso-análisis. Identificar las clases de diseño que realizan dichos requerimientos especiales, si existen. Después, identificar los subsistemas del diseño que contienen a esas clases.

Descripción de interfaces entre subsistemas

Cuando tenemos un esbozo de los subsistemas necesarios para realizar el caso de uso, debemos describir cómo interactúan los objetos de las clases que contiene en el nivel de subsistema. Lo haremos mediante el diagrama de secuencia, que contiene instancias de actores, subsistemas, y transmisiones de mensajes entre éstos.

Captura de requerimientos especiales

Se capturan los requerimientos que se tratarán en la implementación.



Diseño de una clase (por ing. de componentes)

El propósito de diseñar una clase es crear una clase de diseño que cumpla su papel en las realizaciones de los casos de uso y los requerimientos no funcionales que se aplican a estos.

Esbozar una clase

Esbozar clases de diseño tomando como entrada clases de análisis (entidades, de control) y/o interfaces.

Identificar operaciones

Identificamos las operaciones que las clases del diseño van a necesitar y describimos esas operaciones utilizando la sintaxis los lenguajes de programación, incluyendo la visibilidad (public, private, protected, etc.).

Identificar atributos

Identificamos los atributos requeridos por la clase de diseño y los describimos utilizando la sintaxis del lenguaje de programación. Un atributo especifica una propiedad de una clase de diseño y está a menudo implicado y es requerido por las operaciones de la clase.

Identificar asociaciones y agregaciones

Como los objetos del diseño interactúan unos con otros, estas interacciones necesitan asociaciones entre las clases correspondientes.

Identificar las generalizaciones

Las generalizaciones deben utilizarse con la misma semántica definida en el lenguaje de programación.

Describir los métodos

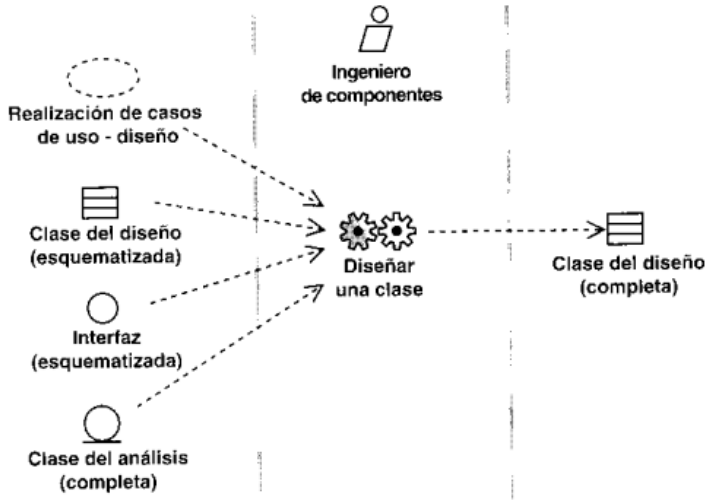
Los métodos pueden ser utilizados en el diseño para especificar como se deben realizar las operaciones. El método puede ser especificado mediante lenguaje natural o pseudocódigo, aunque la mayoría de las veces no son especificados en el diseño, sino en la implementación utilizando directamente el lenguaje de programación.

Describir estados

Se utilizan diagramas de estado para describir las diferentes transiciones de estado de un objeto del diseño.

Tratar requerimientos especiales

En esta fase debe ser tratado cualquier requerimiento especial que no haya sido considerado en pasos anteriores.



Diseño de un subsistema (por ing. de componentes)

Los objetivos del diseño de un subsistema son:

- Garantizar que el subsistema es tan independiente como se posible de otros subsistemas.
- Garantizar que el subsistema proporciona las interfaces correctas.
- Garantizar que el subsistema cumple su propósito de ofrecer una realización correcta de las operaciones tal y como se definen en las interfaces que proporciona.

Mantenimiento de las dependencias entre subsistemas

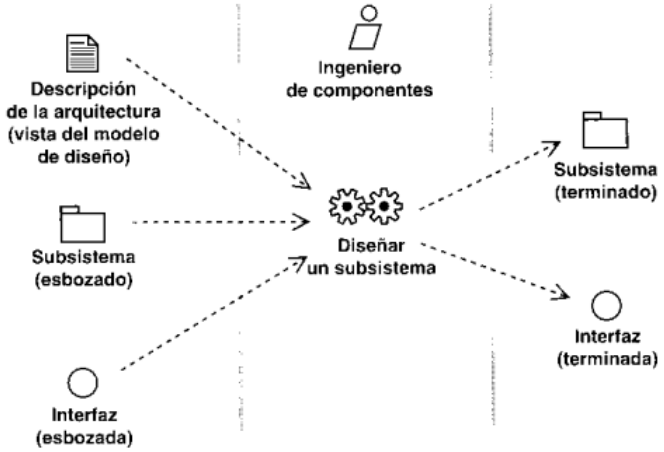
Deberían definirse y mantenerse dependencias en un subsistema respecto a otros cuando los elementos no contenidos en estos últimos estén asociados con elementos dentro de aquel. Hay que minimizar las dependencias entre subsistemas y/o interfaces.

Mantenimiento de interfaces proporcionadas por el subsistema

Las operaciones definidas por las interfaces que proporciona un subsistema deberá soportar todos los roles que cumple el subsistema en las diferentes realizaciones de caso de uso.

Mantenimiento de los contenidos de los subsistemas

Un subsistema cumple sus objetivos cuando ofrece una realización correcta de las operaciones tal y como están descritas por las interfaces que proporciona.



DISEÑO

¿Qué se diseña?

1. Arquitectura.
2. Datos (Bases de datos).
3. Procesos.
4. Interfaces.
5. Formas de entrada/salida.
6. Procedimientos manuales.

Diseño arquitectónico

Arquitectura: es el conjunto de decisiones significativas que tomamos para cumplir los requerimientos del producto. Se muestra a través de vistas (diagramas de UML).

Define la relación entre los principales elementos estructurales del programa.

Toma los requerimientos no funcionales y los aplica al modelo de análisis.

Es un modelo genérico que no entra en detalles.

Diseño de datos (bases de datos)

Transforma los requerimientos en las estructuras de datos necesarias para implementar el software.

Para la persistencia de los objetos, se presenta el problema de que las bases de datos utilizan el modelo relacional. Para ello hay que hacer un mapeo, que se encargará de transformar objetos a tablas de la base de datos y viceversa. A esto se lo conoce como ORM (Object-Relational Mapping).

Diseño de procesos

Transforma elementos estructurales de la arquitectura del programa en una descripción procedimental de los componentes del software.

Se toman en cuenta aspectos como la concurrencia, distribución de procesos, distribución de bases de datos, performance, persistencia, etc.

Diseño de interfaces

Describe como se comunica el software consigo mismo, con los sistemas que operan en él, con los dispositivos externos y con los usuarios. Como se comunica el sistema con el ambiente.

- **Diseño de interfaces externas**
- **Diseño de interfaces de usuario**: son difíciles de diseñar ya que se deben adecuar al conocimiento de las personas que la van a utilizar, y deben ser de utilizada (usabilidad).
- **Diseño de interfaces internas**: un subsistema proporciona un servicio y otro pide el servicio. Son como las interfaces de Java (u otro lenguaje de programación orientado a objetos).

Diseño de formas de entrada/salida

Describe cómo se ingresa información al software y como se presentan las salidas del mismo.

Tipos de procesamiento:

- **En lotes**: la transacción ocurrió, pero el software la procesa después. Se acumulan transacciones para procesarlas después. Se utiliza para procesar procesos lentos cuando el procesador no está ocupado.
- **En línea**: la transacción se procesa en el momento en el que ocurre.
- **En tiempo real**: son sistemas en línea, pero puede modificar el ambiente donde está inmerso. Es muy estricto a los tiempos de respuesta (muy preciso; si un proceso debe durar un determinado tiempo, no puede durar más). Ejemplo: los aspersores que se activan cuando hay un incendio y quitan el oxígeno dentro de 30 segundos.

Diseño de procedimientos manuales

Describe como integra el software al sistema de negocio.

Los procedimientos manuales nos permiten "insertar" el software en el negocio. Ejemplo: capacitación, planes B, etc.

Ejemplo de Plan B: si la página de inscripción a exámenes no funciona, la inscripción se realiza personalmente en bedelía.

Guía para evaluar un buen diseño

- El diseño deberá implementar todos los requerimientos explícitos del modelo de análisis y deberán ajustarse a todos los requerimientos que desea el cliente.
- El diseño deberá ser una guía legible y comprensible para aquellas que generan código y para aquellos que comprueban y consecuentemente, dan soporte al software.
- El diseño deberá proporcionar una imagen completa del software, enfrentándose a los dominios de comportamiento, funcionales y de datos desde una perspectiva de implementación.

Directrices sobre calidad del diseño

- Un diseño deberá presentar una estructura arquitectónica que:
  - Se haya creado mediante patrones de diseño reconocibles.
  - Que esté formada por componentes que exhiban características de buen diseño.
  - Que se puedan implementar de manera evolutiva, facilitando así la implementación y la comprobación.
- Un diseño deberá ser modular: es decir el software deberá dividirse lógicamente en elementos que realicen funciones y sub-funciones específicas.
- Un diseño deberá conducir a interfaces que reduzcan la complejidad de las conexiones entre los módulos y con el entorno externo.

- Un diseño deberá derivarse mediante un mediante un método repetitivo y controlado, de la información obtenida durante el análisis de los requerimientos del software.
- Un diseño deberá contener distintas representaciones de datos, arquitecturas, interfaces y componentes.
- Un diseño deberá conducir a estructuras de datos adecuadas para los objetos que se van a implementar y que procedan de patrones de datos reconocibles.
- Un diseño deberá conducir a componentes que presenten características funcionales independientes.

Principios de diseño del software

- El diseño deberá poderse rastrear hasta el modelo de análisis.
- El diseño no deberá inventar nada que ya esté inventado.
- El diseño deberá "minimizar la distancia intelectual" entre el software y el problema como si de la misma vida real se tratara.
- El diseño deberá presentar uniformidad e integración.
- El diseño deberá estructurase para admitir cambios.

DISEÑO ARQUITECTÓNICO

Introducción

El diseño arquitectónico, es la asignación de modelos de requerimiento esenciales a una tecnología específica. Es el plan del diseño (es la primera etapa del diseño).

El diseño arquitectónico se interesa por entender cómo debe organizarse un sistema y como tiene que diseñarse la estructura global de ese sistema.

Es el enlace crucial entre el diseño y la ingeniería de requerimientos, ya que identifica los principales componentes estructurales en un sistema y su relación entre ellos. La salida del proceso de diseño arquitectónico consiste en un modelo arquitectónico que describe la forma en que se organiza el sistema como un conjunto de componentes comunicados entre sí.

Las arquitecturas de software se diseñan en dos niveles de abstracción:

- **Arquitectura en pequeño:** se interesa por la arquitectura de programas individuales. En este nivel, uno se preocupa por la forma en que el programa individual se separa en componentes.
- **Arquitectura en grande:** se interesa por la arquitectura de sistemas empresariales complejos que incluyen otros sistemas, programas y componentes de programa.

La arquitectura captura la estructura del sistema en términos de componentes y como estos interactúan.

Rol del arquitecto

- Revisar y negociar los requerimientos.
- Documentar la arquitectura.
- Comunicar la arquitectura, asegurando que los interesados la comprendan.
- Direccionar los requerimientos no funcionales a la arquitectura.
- Configurar la arquitectura de hardware.
- Trabajar con el Administrador del Proyecto, ayudando en la planificación, la estimación, la distribución de las tareas y la calendarización del proyecto.

Ventajas de diseñar y documentar la arquitectura

- **Comunicación con los participantes:** la arquitectura es una presentación que sirve para usarse como enfoque para la discusión de un amplio número de participantes.

- **Análisis del sistema:** sirve para aclarar la arquitectura del sistema, para saber si se cumplen los requerimientos críticos.
- **Reutilización a gran escala:** es posible desarrollar arquitecturas de línea de productos donde la misma arquitectura se reutilice mediante una amplia gama de sistemas relacionados.

Requerimientos no funcionales

Los requerimientos no funcionales definen las cualidades o atributos que deben estar presentes en el producto de software resultante.

- Juegan un papel crucial en el diseño y desarrollo del sistema de información.
- Pueden definirse como consideraciones o restricciones asociadas a un servicio del sistema.
- Suelen llamarse también requerimientos de calidad o no comportamentales en contraste con los comportamentales.
- Pueden ser críticos con los funcionales.

Dificultades asociadas a los requerimientos no funcionales

- No hay reglas ni lineamientos para determinar cuándo se obtuvo una solución óptima.
- Tiene buenas y malas soluciones, no soluciones correctas e incorrectas.
- Problemas relacionados con requerimientos no funcionales pueden ser síntomas de problemas mayores.
- Hay dos objetivos que deben poseer: deben ser objetivos y testeables.

Áreas de requerimientos no funcionales

- **Restricciones técnicas:** restringen opciones de diseño para especificar ciertas tecnologías que la aplicación debe usar. Ejemplo: *"solo tenemos diseñadores Java", "la base de datos debe correr con Windows XP"*. Usualmente no son negociables.
- **Restricciones de negocio:** también restringen las opciones de diseño pero por razones de negocio, no por razones técnicas. Ejemplo: *"las licencias son muy costosas, nos vamos a una versión de código abierto", "hay que desarrollar interfaz con el producto X que ya existe en la empresa".* La mayoría de las veces no son negociables.
- **Atributos de calidad del producto:** definen los requerimientos de la aplicación. Los atributos de calidad direccionan aspectos de interés para los usuarios de la aplicación, como así también de otros involucrados. Ejemplo: escalabilidad, disponibilidad, portabilidad, usabilidad, performance, etc.

Clasificación completa de requerimientos no funcionales

- Restricciones técnicas
  - De interoperatividad
  - De implementación
- Restricciones de negocio
  - De estándares
  - De entrega
  - Éticos
  - Legales
- De producto
  - Disponibilidad
  - Performance
    - Concurrencia
    - Uso de recursos
    - Tiempo de respuesta
  - Usabilidad
  - Seguridad
    - Lógica

        - Física
    - Confiabilidad
    - Portabilidad
    - Interfaz
        - Hardware
        - Software
        - Comunicación
        - Usuario

Atributos de calidad del producto (ISO9126)

Los atributos de calidad son parte de los requerimientos no funcionales de una aplicación, que capturan muchas facetas sobre como los requerimientos funcionales son llevados a cabo.

1. **Performance (desempeño)**: establece en cuanto tiempo debe realizarse un trabajo. Hace referencia al rendimiento, tiempo de respuesta y plazos (*deadlines*).
2. **Escalabilidad**: indica cuan preparado está el sistema para crecer sin perder la calidad y estabilidad en los servicios que ofrece. Incluye solicitud de carga, conexiones simultáneas, tamaño de los datos, despliegue.
3. **Modificabilidad**: es una manera de medir que tan fácil es aplicar cambios en la aplicación.
4. **Seguridad**: hace referencia a la autenticación, autorización, encriptación, integridad.
    - Autenticación: la aplicación tiene que poder verificar la identidad de sus usuarios.
    - Autorización: los usuarios y/o roles tienen ciertos privilegios.
    - Encriptación: los mensajes enviados son encriptados.
    - Integridad: asegurar que el contenido de los mensajes no es alterado.
5. **Disponibilidad**: relacionado con la confiabilidad, que pueda ser usado cuando se lo necesite.
6. **Integración:** que la aplicación pueda ser incorporada en un contexto más amplio**.**
7. **Portabilidad**: que una aplicación pueda ser ejecutada en diferentes plataformas de hardware/software. Depende de la tecnología utilizada.
8. **Testeabilidad (capacidad de prueba):** que sea fácil de probar.

Modelado de la arquitectura

- El modelo arquitectónico mapea los requerimientos funcionales del análisis a una arquitectura tecnológica.
- Debe tratar con los requerimientos no funcionales.
- No existe la solución “perfecta”, el modelado arquitectónico debe escoger la solución óptima para el conjunto de circunstancias existentes.
- Debe considerar información sobre:
    - Volúmenes de datos
    - Funcionalidad más demandada del negocio
    - Distribución geográfica
    - Distribución de procesamiento de datos
    - Dónde se guardarán los datos
    - Cuáles procesos se ejecutarán en qué procesadores y que tanta comunicación se requerirá entre ellos.

Salidas del modelo arquitectónico

- Distribución geográfica de los requerimientos de computación.
- Componentes de hardware para las máquinas clientes y para los servidores.
- Configuración y cantidad de niveles de hardware.
- Mecanismos y lenguajes de comunicación de la red.
- Sistema operativo.
- Paradigma de desarrollo
- Lenguaje de presentación (front-end)

- Lenguaje de fondo (back-end)
- Sistema de administración de base de datos
- Ubicación/es de los procesos
- Ubicación/es de los datos físicos
- Estrategias de sincronización de los datos distribuidos físicamente

Elección del modelo arquitectónico y los requerimientos no funcionales

Aunque cada sistema es único, los sistemas en el mismo dominio de aplicación tienen normalmente arquitecturas similares que reflejan los conceptos fundamentales del dominio. Cuando se diseña una arquitectura de sistema, debe decidirse que tienen en común el sistema y las clases de aplicación más amplias, con la finalidad de determinar cuánto conocimiento se puede reutilizar de dichas arquitecturas de aplicación.

La arquitectura de un sistema de software puede basarse en un patrón o en un estilo arquitectónico particular. Un patrón arquitectónico es una descripción de una organización del sistema. Éstos captan la esencia de una arquitectura que se usó en diferentes sistemas de software.

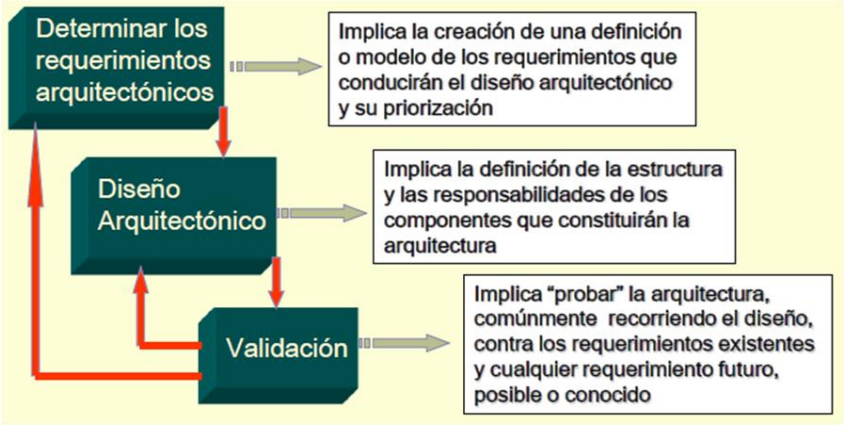
Debido a la estrecha relación entre los requerimientos no funcionales y la arquitectura de software, el estilo y la estructura arquitectónica particulares que se elijan para un sistema dependerán de los requerimientos no funcionales del sistema:

- **Desempeño (performance)**: si es un requerimiento crítico, esto sugiere que la arquitectura se debe diseñar para localizar operaciones críticas dentro de un número reducido de subsistemas con poca comunicación (hasta donde sea posible) entre estos subsistemas, es decir, componentes de granulidad gruesa para reducir componentes de comunicación.
- **Seguridad**: si es un requerimiento crítico, esto sugiere utilizar una estructura en capas para la arquitectura con los recursos más críticos protegidos con capas internas y un alto nivel de validación aplicado a esas capas.
- **Protección**: si es un requerimiento crítico, sugiere que la arquitectura debe diseñarse para incluir operaciones relacionadas con la protección se localicen en un solo subsistema o en un número reducido de subsistemas. Esto reduce costos y problemas de validación.
- **Disponibilidad**: si es un requerimiento crítico, debe diseñarse una arquitectura que incluya componentes redundantes de tal forma que puedan reemplazarse y actualizarse sin detener el sistema.
- **Mantenibilidad**: si es un requerimiento crítico, la arquitectura debe diseñarse utilizando componentes autocontenidos de granularidad fina que puedan cambiarse con facilidad. Los productores de datos separados de los consumidores y las estructuras de datos compartidas deben evitarse.

Conflictos arquitectónicos

- Utilizar componentes de granulidad alta mejora la performance pero reduce la mantenibilidad.
- La introducción de datos redundantes mejora la disponibilidad pero hace la seguridad más difícil.
- La localización de aspectos de seguridad relacionados usualmente significa más comunicación, por lo tanto degrada la performance.

Proceso de arquitectura de software



1) Determinar los requerimientos arquitectónicos

Antes de que una solución arquitectónica sea diseñada, es necesario tener una buena idea de los requerimientos para la arquitectura de la aplicación. Los requerimientos arquitectónicos son esencialmente la calidad y los requerimientos no funcionales de la aplicación.

Identificar los requerimientos arquitectónicos

Se toman como entrada los requerimientos funcionales y los requerimientos de los involucrados (*stakeholders*). Se determinan los requerimientos arquitectónicos, y se produce como salida un documento que contiene los requerimientos arquitectónicos de la aplicación.

Priorizar requerimientos

Los requerimientos se ubican en tres categorías:

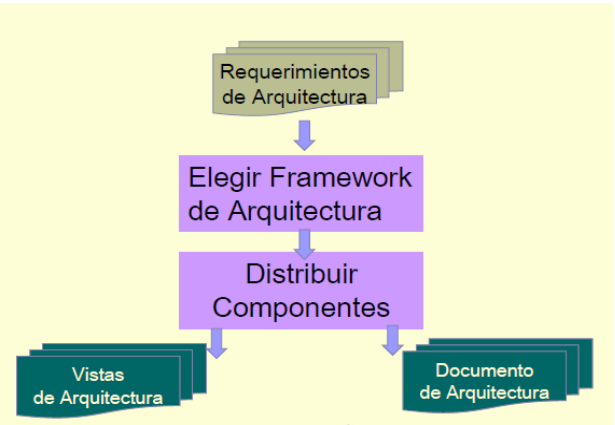
- **Alto**: la aplicación debe soportar este requerimiento. Conducen el diseño de la arquitectura.
- **Medio**: necesitará ser soportado en alguna etapa, pero no necesariamente en el primer release.
- **Bajo**: estos requerimientos son parte de la lista de deseos. Las soluciones que los incluyen son deseables pero no conducen el diseño.

2) Proceso de diseño arquitectónico

Elección del framework de la arquitectura

La mayoría de las aplicaciones están basadas en un pequeño número de arquitecturas probadas. Hay una buena razón para esto, funcionan. Utilizar estas soluciones ya probadas hace que los riesgos se minimicen.

No hay una fórmula mágica para diseñar un framework de arquitectura. Un pre-requisito es conocer los patrones arquitectónicos para abordar ciertos atributos de calidad.



Distribuir componentes

Una vez seleccionado el framework arquitectónico, basado en uno o más patrones arquitectónicos, la siguiente tarea es definir los componentes que comprenderán el diseño.

Para ello se debe identificar:

- Los componentes principales de la aplicación**.**
- Servicios o interfaces que cada componente soporta**.**
- Responsabilidades del componente**.**

- Dependencias entre componentes**.**
- Particiones en la arquitectura que son candidatas para distribución entre servidores de red.

Lineamientos para el diseño de componentes:

- Minimizar dependencias entre componentes
  - Recordar: si lo cambias, debes re-testearlo.
- Diseñar componentes que encapsulen un conjunto de responsabilidades altamente cohesivas**.**
- Aislar dependencias entre middleware y cualquier COT de infraestructura tecnológica**.**
  - Es más fácil de construir, pero introduce penalidades de performance**.**
- Utilizar la descomposición para estructurar componentes jerárquicamente
- Minimizar las llamadas entre componentes.

3) Validación

- Es "probar" la arquitectura.
- Ayuda a incrementar la confianza del equipo de diseño en que la arquitectura cumpla su propósito.
- Debe realizarse con las restricciones de tiempo y presupuesto del proyecto.
- Es un desafío validar la arquitectura, ya que finalmente es un diseño y no puede ejecutarse ni probarse para ver si cumple con los requerimientos.

Hay dos técnicas principales para la validación:

- **Uso de escenarios**: consiste en un testeo manual de la arquitectura usando casos de prueba.
- **Prototipos**: implica la construcción de un prototipo que crea un arquetipo simple de la aplicación deseada. Tiene la capacidad de evaluar los requerimientos en forma detallada. Los prototipos son versiones mínimas, restringidas de la aplicación creados específicamente para probar algún aspecto riguroso o pobremente comprendido en el diseño. Son utilizados para dos propósitos:
  - Prueba de conceptos.
  - Prueba de tecnología.

El objetivo de las dos técnicas es identificar potenciales defectos y debilidades en el diseño para que puedan ser mejoradas antes de la fase de implementación.

Documentar la arquitectura

¿Para qué se usa?

- Expresión del sistema y su evolución.
- Comunicación entre los involucrados.
- Evaluación y comparación de la arquitectura de una forma consistente.
- Planificación, administración y ejecución de las actividades del desarrollo del sistema.
- Expresión de las características persistentes y soporte de los principios que guían un cambio aceptable.
- Verificación de la implementación del sistema.

PATRONES ARQUITECTÓNICOS

Introducción

Un patrón arquitectónico se puede considerar como una descripción abstracta estilizada de buena práctica, que se ensayó y puso a prueba en diferentes sistemas y entornos. Un patrón arquitectónico debe describir una organización de sistema que ha tenido éxito en sistemas previos. Debe incluir información sobre cuándo es y cuándo no es adecuado usar dicho patrón, así como las fortalezas y debilidades del patrón.

Existen patrones para distintos niveles de solución:

- **Patrones de bajo nivel y detallados:** idiomas (lenguajes de programación).
- **Patrones de nivel medio:** patrones de diseño (objetos comunes y patrones de clases).
- **Patrones de alto nivel:** patrones arquitectónicos (componentes y módulos).

Patrones arquitectónicos vs. Estilos arquitectónicos

Suele ser difícil distinguirlos. A veces se los ve como sinónimos.

- **Patrón**: los patrones están a una escala menor que los estilos. Múltiples patrones pueden aparecer en un diseño.
- **Estilo**: en contraste, un sistema usualmente, tiene un único estilo arquitectónico dominante.

Platónicos vs. Embebidos

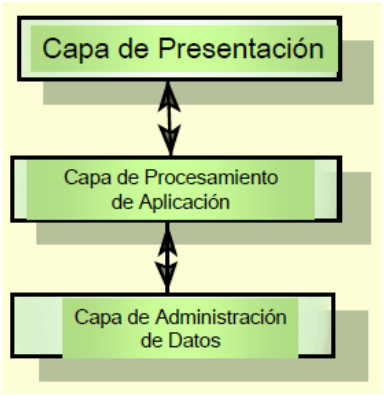
- **Platónicos:** son patrones idealizados, los que se ven en los libros y rara vez de igual forma en el código.
- **Embebidos:** se los ve en sistemas reales y suelen violar las restricciones estrictas de los platónicos. Esta violación implica una gran compensación.

Patrones

1. *Layered* (estructura en capas).
2. Repositorio (o modelo centrado, *Repository*).
3. Modelo Vista Controlador (MVC, *Model View Controller*).
4. Publicar y Suscribir (*Publish-Suscribe*).
5. *Messaging*.
6. Tubería y Filtro (*Pipe & Filter*).
7. Lote-Secuencial (*Batch-Sequential*).
8. Agente (*Broker*).
9. Coordinador de Proceso (*Process Coordinator*).

Layered (capas o estratificado)

Consiste en una pila de capas, donde cada capa actúa como una máquina virtual de la capa de arriba.



En una forma de lograr la separación y la independencia. Aquí, la funcionalidad del sistema está organizada en capas separadas, y cada capa se apoya solo en las facilidades y los servicios ofrecidos por la capa inmediatamente debajo de ella (las capas solo pueden usar a las capas que se encuentran directamente debajo de ellas).

Calidad resultante: trata directamente los atributos de modificabilidad, portabilidad y reusabilidad. Mientras su interfaz no varíe, una capa puede sustituirse por otra equivalente; o cuando las interfaces de capa cambian o se agregan nuevas facilidades a una capa, sólo resulta afectada la capa adyacente.

Variantes:

- Algunas evitan la restricción de modo que las capas puedan comunicarse con capas de más abajo.
- Otra variante es el uso de capas compartidas, donde cada capa puede usar estas capas verticales.

Notas: el estilo estratificado puede variar considerablemente en su forma platónica de su forma embebida. En la práctica se pueden saltar capas hacia capas inferiores, lo cual provoca negar los atributos de calidad que son su beneficio. Aun así es beneficiosa dado que las capas agrupan módulos de funcionalidad coherente.

Descripción	Organiza el sistema en capas con funcionalidad relacionada con cada capa. Una capa da servicios a la capa de encima.
Cuando se usa	Se usa al construirse nuevas facilidades encima de los sistemas existentes, cuando el desarrollo se dispersa a través de varios equipos de trabajo, cuando existe un requerimiento de capa multinivel.
Ventajas	Permite la sustitución de capas completas (mientras se conserve la interfaz).
Desventajas	En la práctica suele ser difícil ofrecer una separación limpia entre capas, donde posiblemente una capa de nivel superior tenga que interactuar con capas del nivel inferior directamente. El rendimiento suele ser un problema debido a los múltiples niveles.

Repositorio (modelo centrado)

Consiste en un conjunto de componentes independientes que interactúan con un modelo central (llamado también almacenamiento de datos o repositorio) en vez de interactuar unos con otros.

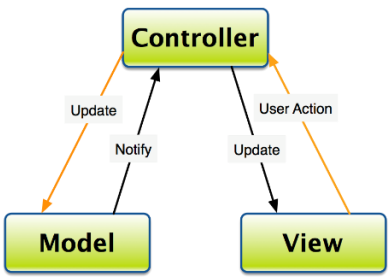
Está relacionado con el patrón MVC.

Calidad resultante: los sistemas repositorio son altamente modificables, debido a la independencia de los componentes; son extensibles, ya que es muy fácil agregar componentes.

Descripción	Todos los datos de un sistema se gestionan en un repositorio central, accesible a todos los componentes del sistema. Los componentes no interactúan directamente, sino tan sólo a través del repositorio.
Cuando se usa	Se usa cuando se tiene un sistema donde los grandes volúmenes de información generados deban almacenarse durante mucho tiempo. Puede usarse también en sistemas dirigidos por datos, en los que la inclusión de datos en el repositorio active alguna acción.
Ventajas	Los componentes pueden ser independientes, no necesitan conocer la existencia de otros componentes. Los cambios hechos por un componente se pueden propagar hacia todos los componentes. La totalidad de datos se puede gestionar de una manera consistente (todo está en un solo lugar). Es extensible, se pueden agregar vistas y controladores más tarde.
Desventajas	El repositorio es un punto de falla único (los problemas en el repositorio afectan a todo el sistema). Puede ser ineficiente la organización de la comunicación a través del repositorio. Difícil de distribuir en forma eficiente.

MVC (*Model View Controller*)

Descripción	Separa presentación e interacción de los datos del sistema. El sistema se estructura en tres componentes lógicos que interactúan entre sí. El modelo maneja los datos del sistema y las operaciones asociadas a esos datos. La vista define y gestiona como se presentaran los datos al usuario (refleja el estado del modelo). El controlador dirige la interacción del usuario y pasa esas interacciones a vista y modelo (recibe peticiones de la vista sobre el modelo, interacciona con el modelo, devuelve el resultado a la vista).
Cuando se usa	Se usa cuándo existen múltiples formas de interactuar con los datos. También se utiliza cuando se desconocen los requerimientos futuros para la interacción y presentación.
Ventajas	Permite que los datos cambien de manera independiente de su representación y viceversa. Soporta diferentes representaciones de los mismos datos, y los cambios en una representación se muestra en todos ellos. *Incrementa flexibilidad y reutilización.*
Desventajas	Puede implicar código adicional y complejidad de código cuándo el modelo de datos y las interacciones son simples.



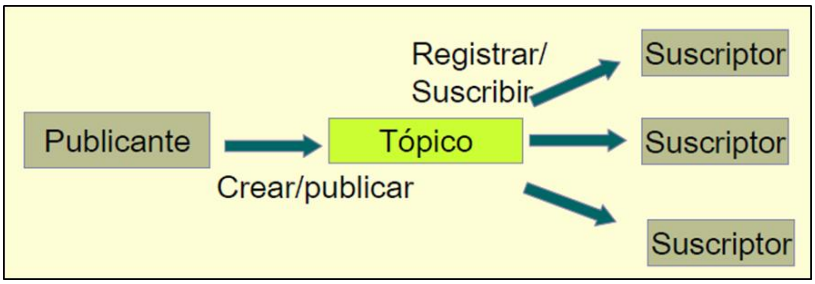
Publicar y Suscribir (*Publish-Suscribe*)

Consiste en componentes que publican eventos y otros se suscriben a ellos.

Los publicantes ignoran el motivo global por el cual el evento es publicado, los suscriptores ignoran porque o quien publica el evento y dependen sólo del evento, no de quién lo publica. A su vez, los publicantes son inconscientes del consumo de los eventos.

Cada tópico puede tener más de un publicante y los publicantes pueden aparecer y desaparecer dinámicamente, lo que le da flexibilidad sobre configuraciones estáticas.

Los suscriptores pueden suscribirse y des-suscribirse dinámicamente a un tópico.



Mensajería muchos a muchos: los mensajes publicados son enviados a todos los suscriptores que están registrados en el tópico. Muchos publicantes pueden publicar en el mismo tópico, y muchos suscriptores pueden escuchar en el mismo tópico.

Los buses de eventos varían en las propiedades que soportan:

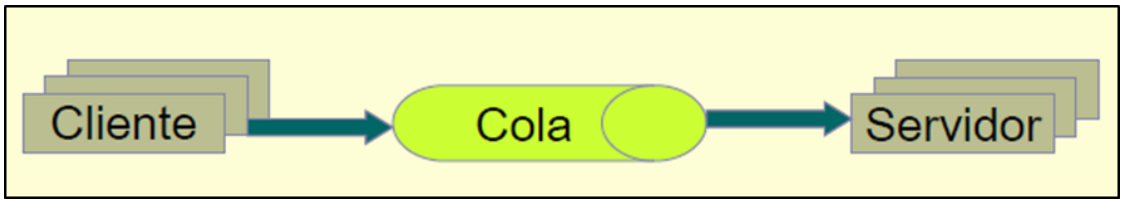
- **Durables**: garantizan que cualquier mensaje que aceptan no se perderá durante una falla (los eventos se escriben en un almacenamiento confiable).
- **Entrega en orden**: garantizan la entrega en orden o priorizada de eventos.
- **Entrega en lotes**: los eventos se acumulan en lotes para evitar cúmulos de eventos similares.

Descripción	Componentes independientes publican eventos y otros se suscriben a ellos.
Cuando se usa	En arquitecturas muy flexibles y adecuadas para aplicaciones que requieren mensajes asíncronos uno-a-muchos, muchos-a-uno, muchos-a-muchos componentes.
Ventajas	No hay vínculo directo entre publicantes y suscriptores. Los publicantes no saben quién recibe su mensaje, y los suscriptores no saben qué publicante envía el mensaje. Desacopla los componentes que producen los eventos de quienes los consumen, haciendo que la arquitectura sea mantenible y evolucionable.
Desventajas	El bus de eventos agrega una capa de indirección entre productores y consumidores, pudiendo afectar la performance del sistema.

Messaging

Esta arquitectura desacopla emisores de receptores utilizando una cola de mensajes intermedia.

El emisor envía un mensaje al receptor y sabe que eventualmente será entregado, aunque la red esté caída o el receptor no esté disponible.



Propiedades claves:

- **Comunicaciones asíncronas**: clientes envían pedidos a una cola, donde se almacenan hasta que una aplicación los remueve, luego el cliente continúa sin esperar la respuesta.
- **Bajo acoplamiento**: no hay vínculo directo entre emisores y receptores. El cliente es inconsciente de cual servidor recibe el mensaje y, el servidor es inconsciente del cliente que envía el mensaje.

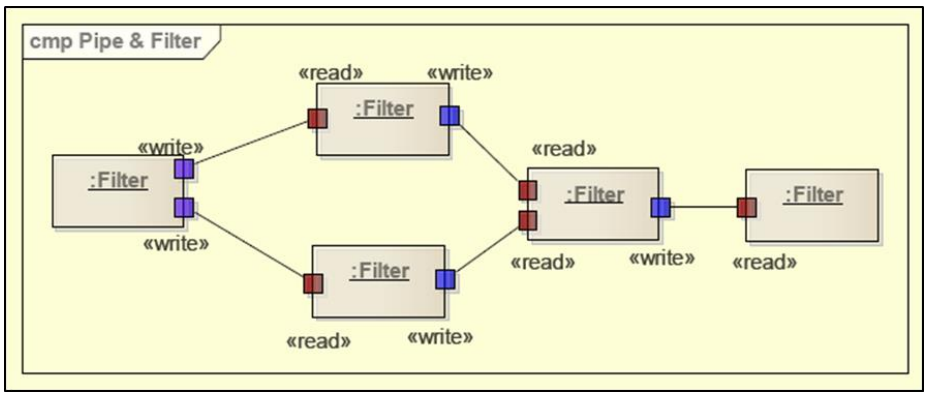
Descripción	Emisores envían mensajes a receptores, donde los mensajes son almacenados en una cola.
Cuando se usa	Provee una solución resistente para aplicaciones en las cuales la conectividad es transitoria, tanto debido a la red como a la poca confiabilidad de los servidores. Es apropiada cuando el cliente no necesita una respuesta inmediata después de enviar el mensaje.
Ventajas	Promueve el bajo acoplamiento, permitiendo alta modificabilidad, dado que emisores y receptores no se conectan.
Desventajas	El bus de eventos agrega una capa de indirección entre productores y consumidores, pudiendo afectar la performance del sistema.

Tubería y filtro (*Pipe & Filter*)

Los datos fluyen de un componente a otro (filtros), a través de una tubería, y se transforman conforme se desplazan en la secuencia. Cada paso de procesamiento se implementa como un transformador. Los datos de entrada fluyen por medio de dichos transformadores hasta que se convierten en salida.

Una característica clave es que toda la red de tuberías está continua e incrementalmente procesando datos.

Descripción	Los datos fluyen (como en una tubería) de un componente a otro para su procesamiento. El procesamiento de datos se organiza de forma que cada componente de procesamiento (filtro) sea discreto y realice un tipo de transformación de datos.
Cuando se usa	Se suele utilizar en aplicaciones de procesamiento de datos (ya sean Batch –en lotes- o en transacciones), donde las entradas se procesan en etapas separadas para generar salidas relacionadas.
Ventajas	Es fácil de entender y soporta reutilización de transformación. El estilo de flujo de trabajo coincide con la estructura de muchos procesos empresariales. La evolución al agregar transformaciones es directa. Puede implementarse como sistema secuencial o como uno concurrente.
Desventajas	El formato para transferencia de datos debe acordarse entre las transformaciones que se comunican. Cada transformación debe analizar sus entradas y sintetizar sus salidas al formato acordado. Esto aumenta la carga del sistema y dificulta la reutilización de transformaciones funcionales que usen estructuras de datos compatibles. Son inapropiadas para aplicaciones interactivas.

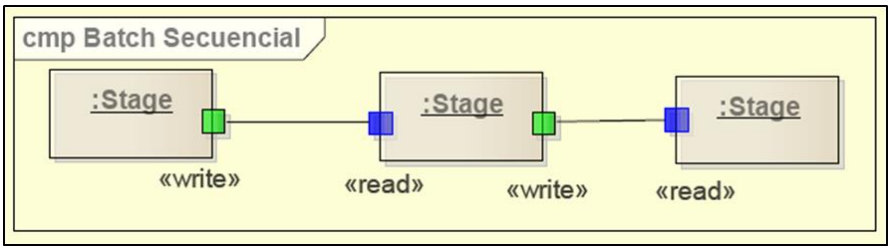


Lote-Secuencial (*Batch-Sequential*)

Los datos fluyen de una etapa a otra y es procesado incrementalmente.

A diferencia del Pipe & Filter, cada etapa completa todo su procesamiento antes de escribir su salida.

Los datos fluyen entre etapas en una cadena, pero es más común escribir en un archivo o disco.



Tiene restricciones similares al Pipe & Filter:

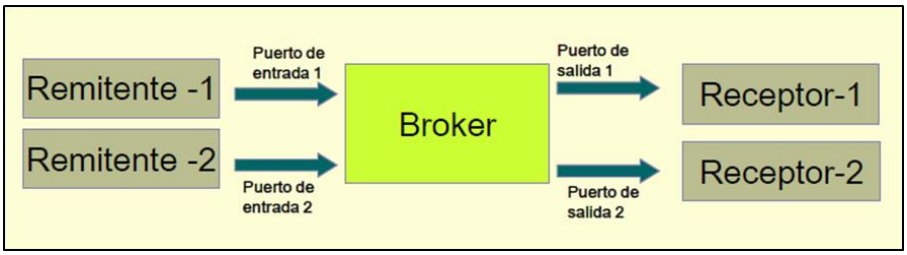
- Cada etapa o paso es independiente y depende de los datos que toma, no de la etapa anterior.
- Las etapas no interactúan entre sí excepto a través de la cadenas de entradas o salidas de archivos.
- Es una serie de pasos lineales.
- No se hace trabajo en los conectores, salvo el paso de datos.

Descripción	Los datos fluyen de una etapa a otra y es procesado incrementalmente.
Ventajas	Alta modificabilidad, ya que cada etapa es independiente de las otras. Buen rendimiento. Conceptualmente simple dado que una etapa está ejecutándose por vez.
Desventajas	La salida final está completamente terminada o ausente, lo que impacta en la usabilidad del sistema (a diferencia del Pipe & Filter). No se puede ejecutar trabajo en paralelo a menos que fluyan múltiples trabajos a través del sistema.

Agente (*Broker*)

Propiedades clave del patrón:

- **Arquitectura Hub-and-spoke**: el agente actúa como concentrador de mensajes y los remitentes y receptores se conectan como rayos. Las conexiones al agente son por medio de puertos asociados a un formato específico de mensaje.
- **Realiza ruteo de mensajes**: el agente incrusta la lógica de procesamiento para entregar un mensaje recibido en un puerto de entrada en el puerto de salida.
- **Realizar transformación de mensajes**: el agente lógico transforma el tipo de mensaje de origen recibido en el puerto de entrada en el tipo de mensaje de destino requerido por el puerto de salida.



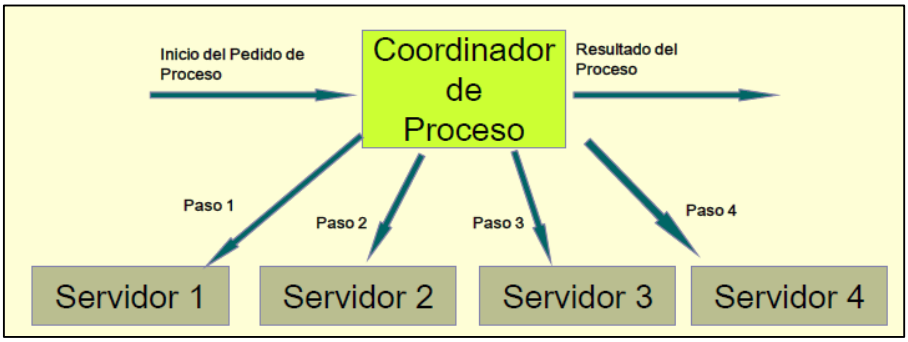
Cuando se usa	Son adecuados para aplicaciones en las cuales los componentes intercambian mensajes que requieren grandes transformaciones entre los formatos de origen y destino.
Ventajas	El agente desacopla el remitente del receptor, permitiéndoles producir o consumir su formato nativo, y centraliza la definición de la transformación lógica en el agente para facilitar la comprensión y modificación.

Coordinador de procesos (*Process Coordintator*)

Propiedades clave del patrón:

- **Encapsulamiento de proceso**: el coordinador de proceso encapsula la secuencia de pasos necesarios para completar un proceso de negocio. Recibe un requerimiento de inicio de proceso, llama a los servidores en el orden definido por el proceso y emite los resultados.

- **Bajo acoplamiento**: los componentes del servidor son inconscientes de su rol en el proceso de negocio general, y del orden de los pasos del proceso. Los servidores simplemente definen un conjunto de servicios que pueden ejecutar y el coordinador los llama cuando es necesario.
- **Comunicaciones flexibles**: las comunicaciones entre el coordinador y los servidores pueden ser sincrónicas o asincrónicas. Para las comunicaciones sincrónicas, el coordinador espera hasta que el servidor responda. Para las comunicaciones asincrónicas, el coordinador provee una llamada o respuesta cola/tópico, y espera hasta que el servidor responde utilizando el mecanismo definido.



Cuando se usa	Es utilizado comúnmente para implementar procesos de negocio complejos que deben publicar requerimientos a algunos componentes servidores diferentes.
Ventajas	Encapsulando la lógica de proceso en un lugar, es más fácil cambiar, administrar y monitorear la performance del proceso.

ARQUITECTURA DE SISTEMAS DISTRIBUIDOS

Tipos de sistemas

- **Sistemas personales**: no son distribuidos y han sido diseñados para ejecutarse en una computadora personal o estación de trabajo.
- **Sistemas embebidos**: se ejecutan en un único procesador o en un grupo integrado de procesadores.
- **Sistemas distribuidos**: el sistema de software se ejecuta en un grupo de procesadores cooperativos integrado conectados por una red.

Introducción

Prácticamente todos los sistemas basados en computadoras son ahora sistemas distribuidos. Un sistema distribuido es aquel que implica numerosas computadoras, en contraste con los sistemas centralizados en que todos los componentes del sistema se ejecutan en una sola computadora. Es una colección de computadoras independientes que aparecen al usuario como un solo sistema coherente.

Un sistema distribuido debe ser transparente. Esto significa que los usuarios ven el sistema como un solo sistema cuyo comportamiento no resulta afectado por la forma en que se distribuye el sistema. En la práctica, es imposible hacer un sistema por completo transparente (demoras en la red, fallas del nodo remoto, etc.).

Características de los sistemas distribuidos

1. **Compartición de recursos**: permiten compartir recursos de hardware y software a través de una red.
2. **Apertura**: son abiertos, lo que significa que se diseñan sobre protocolos estándares que combinan software y equipos de diferentes vendedores.
3. **Concurrencia**: varios procesos pueden están operando al mismo tiempo sobre diferentes computadoras de la red.

4. **Tolerancia a fallas**: la disponibilidad de muchas computadoras y el potencial de reproducir información significa que los sistemas distribuidos pueden tolerar fallas de hardware y software.
5. **Escalabilidad**: los sistemas pueden crecer incrementando recursos para cubrir nuevas demandas. La escalabilidad de un sistema refleja su disponibilidad para entregar una alta calidad de servicio conforme aumentan las demandas del sistema. Hay tres dimensiones de escalabilidad:
    - Tamaño: debe ser posible agregar más recursos.
    - Distribución: dispersar los recursos sin reducir su rendimiento.
    - Manejabilidad: debe poder administrarse un sistema conforme aumenta su tamaño, incluso si partes del sistema se ubican en organizaciones diferentes.

Desventajas de los sistemas distribuidos

1. **Complejidad**: son más difíciles de comprender y probar.
2. **Seguridad**: dificulta asegurar la integridad y la degradación del servicio.
3. **Manejabilidad**: los defectos pueden propagarse de una máquina a otra, lo que implica que es más difícil de gestionar y administrar.
4. **Impredecibilidad**: la respuesta depende de la carga total del sistema, de la organización y de la red.

Ataques de los que deben defenderse los sistemas distribuidos

1. **Intercepción**: un atacante intercepta comunicaciones entre partes del sistema, provocando que haya poca confidencialidad.
2. **Interrupción**: los servicios son atacados y no pueden entregarse como se esperaba. Se bombardea al servicio con peticiones ilegitimas para que no pueda atender las legítimas.
3. **Modificación**: el atacante cambia los datos o el servicio del sistema.
4. **Fabricación**: el atacante genera información que no debe existir y luego la usa para conseguir ciertos privilegios. Además, hace que el sistema maneje información poco confiable o fuera de la realidad.

Modelos de interacción en sistemas distribuidos

- **Interacción procedimental**: una computadora solicita un servicio conocido a otra computadora y espera que la otra computadora le responda. *Sincrónico.*
- **Interacción basada en mensajes**: la computadora emisora define en un mensaje la información y lo envía a otra computadora. No es necesario esperar la respuesta. Normalmente se implementa con un componente que crea un mensaje, el cual detalle los servicios requeridos por otro componente. *Asincrónico.*

Middleware

Es el software que permite gestionar y garantizar que las diversas partes de un sistema distribuido puedan comunicarse e intercambiar datos.

Se llama *middleware* porque se encuentra en el centro de las partes del sistema distribuido.

Por lo general, se implementa como un conjunto de librerías, que se instala en cada computadora distribuida.

El middleware es software de propósito general (enlatado) que se consigue, en lugar de que los desarrolladores de aplicación los escriban especialmente.

Ejemplos: comunicación con bases de datos, gestores de transacciones, convertidores de datos y controladores de comunicación.

En un sistema distribuido brinda dos tipos de soporte:

- **Soporte de interacción**: el middleware coordina las interacciones entre diferentes componentes del sistema.

- **Provisión de servicios comunes**: el middleware proporciona implementaciones de reutilización de servicios que pueden requerir varios componentes en el sistema distribuido.

Patrones arquitectónicos para sistemas distribuidos

Los diseñadores de sistemas distribuidos deben organizar sus diseños de sistema para encontrar un equilibrio entre rendimiento, confiabilidad, seguridad y manejabilidad del sistema. Cuando se diseña una aplicación distribuida, se deberá elegir un estilo arquitectónico que soporte los requerimientos no funcionales críticos del sistema. Cinco estilos arquitectónicos de sistemas distribuidos:

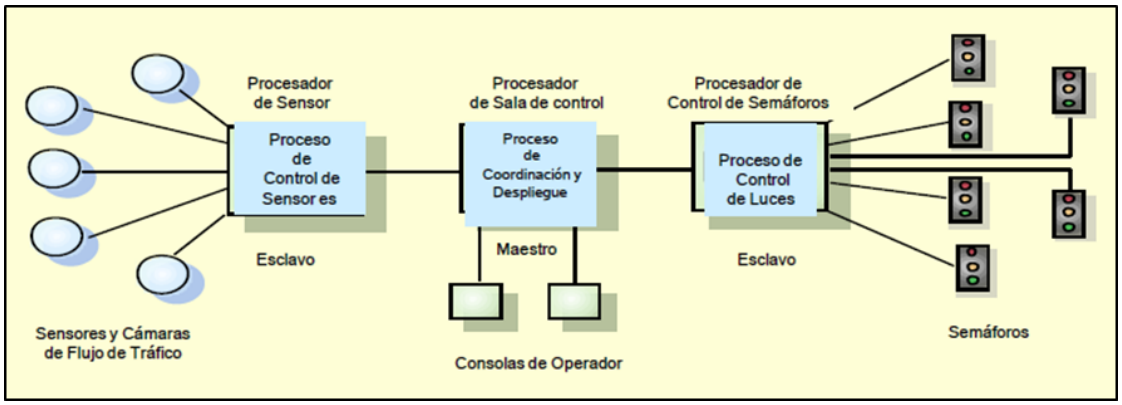
1. Arquitectura maestro-esclavo (*master-slave*)
2. Arquitectura cliente-servidor
    a. De dos niveles
    b. Multinivel
3. Arquitectura de componentes distribuidos
4. Arquitectura *Peer-to-peer*
5. Arquitectura *Map-Reduce*

Arquitectura Maestro-Esclavo (*Master-Slave*)

Se usan en sistemas de tiempo real donde es importante cumplir con los tiempos de procesamiento, donde puede haber procesadores separados para la adquisición de datos del entorno del sistema, procesamiento de datos y actuador de gestión.

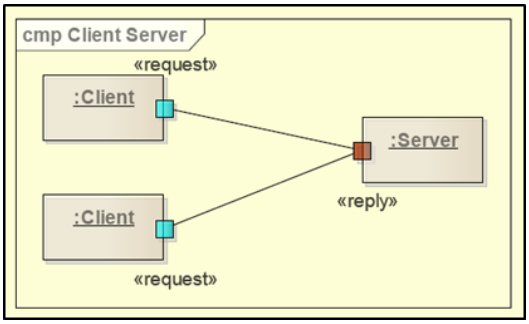
El proceso "maestro" es el responsable de la computación, coordinación, comunicación y control de los procesos "esclavos". Los procesos "esclavos" se dedican a acciones específicas, como adquisición de datos de sensores.

Se usa cuando es posible predecir el procesamiento distribuido que se requiere y cuando se puede asignar fácilmente el procesamiento a los procesadores "esclavos". Esta situación es común en los sistemas de tiempo real.



Arquitecturas cliente-servidor

La aplicación es modelada como un conjunto de servicios que son provistos por servidores y un conjunto de clientes que usan esos servicios. Los clientes piden servicios a los servidores.



Los clientes conocen los servicios y deben saber cómo buscarlos, pero los servidores no necesitan conocer a los clientes.

Los que inician la comunicación son los clientes. Los servidores no conocen la identidad de los clientes hasta que se han contactado.

Los servidores se pueden replicar, esto es útil cuando la carga de un sistema es variable.

Si bien es una arquitectura distribuida, puede implementarse también en una sola computadora.

Los sistemas clientes-servidor dependen de que exista una clara separación entre la presentación de la información y los cálculos que crea y procesa esa información. En consecuencia, se debe diseñar la arquitectura de los sistemas distribuidos cliente-servidor para que se estructuren en varias capas lógicas, con interfaces claras entre dichas capas. Esto permite que cada capa se distribuya en diferentes computadoras.

Variantes:

- Los conectores pueden ser sincrónicos o asincrónicos.
- Puede haber límites en el número de clientes y servidores.
- Las conexiones pueden ser con o sin estado (sesiones).
- Las topologías del sistema pueden ser estática o dinámica.

Arquitectura cliente-servidor de dos niveles

Es la forma más simple de la arquitectura cliente servidor. El sistema se implementa en un solo servidor más un número indefinido de clientes que usan dicho servidor.

- **Modelo cliente ligero**: la capa de presentación se implementa en el cliente, y todas las otras capas (de procesamiento, gestión de datos, etc.) se implementan en el servidor. Puede ser un programa de escritorio, pero es más común usar un navegador web.
  La ventaja del cliente ligero consiste en que es simple de manejar por los clientes. Puede ser costoso instalar y difícil instalar un software en cada uno de ellos si existe un número considerable de clientes. La solución es utilizar un navegador web, haciendo que no sea necesario instalar el software cliente.
  La desventaja de este enfoque es que se puede provocar una fuerte carga de procesamiento en el servidor como en la red, ya que el servidor realiza toda la computación.
- **Modelo cliente pesado**: parte o todo el procesamiento de la aplicación se realiza en el cliente. Utiliza el poder de procesamiento disponible en la máquina cliente. Las funciones de gestión de datos y base de datos se implementan en el servidor.
  La ventaja es que se distribuye el procesamiento, aliviando la carga de procesamiento del servidor y de la red.
  La desventaja es que requiere gestión de sistema adicional para implementar y mantener el software en la computadora del cliente. Una nueva versión debe ser instalada en todos los clientes.

El problema fundamental de este enfoque, es que las capas lógicas del sistema deben mapearse en dos sistemas de cómputo: el cliente y el servidor. Esto puede producir problemas con la escalabilidad y rendimiento si se elige el modelo de cliente ligero, o problemas de gestión de sistema si se usa el modelo de cliente pesado. Para ello se utiliza una arquitectura C/S multinivel.

Arquitectura cliente-servidor multinivel

En esta arquitectura, las diferentes capas del sistema (presentación, gestión de datos, procesamiento de aplicación y base de datos) son procesos separados que pueden ejecutarse en diferentes procesadores.

El modelo cliente-servidor de tres niveles puede extenderse a una variante multinivel, donde se agregan servidores adicionales al sistema. Esto podría implicar el uso de un servidor web para gestión de datos y servidores separados para procesamiento de aplicación (lógica de negocio) y servicios de bases de datos.

Los sistemas cliente-servidor multinivel que distribuyen el procesamiento de aplicación a través de varios servidores son inherentemente más escalables que las arquitecturas de dos niveles.

Arquitectura cliente-servidor N-Tier

Propiedades clave:

- **Separación de intereses**: capas diferentes y claramente divididas para presentación, negocios y manejo de datos.
- **Comunicaciones sincrónicas**: la comunicación entre niveles es pedido-respuesta sincrónica. Las peticiones emanan en una única dirección: del nivel de cliente a los niveles web y de la lógica de negocio al nivel EIS. Cada nivel espera la respuesta del otro nivel antes de proseguir.
- **Distribución flexible**: no hay restricciones para la distribución multi capas de la aplicación. En aplicaciones web, el cliente usualmente corre en un browser de una PC, comunicándose remotamente con internet con componentes a nivel web.

Calidad resultante:

- Mejora la mantenibilidad: soporta cambios en las reglas de negocio cambiando el servidor, en lugar de hacerlo en múltiples clientes.
- Puede enmascarar sistemas legados y tratarlos como servidores.

Arquitectura de componentes distribuidos

Consiste en diseñar el sistema como un conjunto de servicios, sin tratar de asignar dichos servicios a capas en el sistema. Cada servicio, o grupo de servicios relacionados, se implementa usando un componente separado. El sistema está organizado como un conjunto de componentes u objetos en interacción, que proporcionan una interfaz a un conjunto de servicios que ofrecen. Otros componentes solicitan dichos servicios a través de middleware.

Estos sistemas dependen del middleware que gestiona las interacciones entre los componentes. Ejemplo de middleware: CORBA.

Ventajas:

- Permite al diseñador del sistema demorar las decisiones acerca de dónde y cómo deben proporcionarse los servicios.
- Permite adicionar nuevos recursos según se requiera.
- El sistema es flexible y escalable.
- Es posible reconfigurar dinámicamente el sistema con componentes que migran a través de la red.

Desventajas:

- Son más complejos de diseñar que los sistemas cliente-servidor.
- El middleware estandarizado para sistemas de componentes distribuidos nunca se ha aceptado en la comunidad. Ejemplo: Microsoft y Sun han desarrollado middleware diferentes e incompatibles.

Arquitectura *Peer-to-Peer (P2P)*

Los sistemas entre pares (peer-to-peer, p2p) son sistemas descentralizados en los que los cálculos pueden realizarse en cualquier nodo de la red. No se hacen distinciones entre clientes y servidores. En este tipo de sistemas, el sistema global está diseñado para sacar ventaja del poder computacional y de almacenamiento disponible en una red de computadoras potencialmente enorme. Aquí cada nodo, tiene la habilidad (no la obligación) de comportarse como cliente o como servidor. El resultado es un conjunto de nodos operando como pares donde cada nodo puede pedir o proveer servicios a cualquier otro nodo de la red.

Calidad resultante:

- Disponibilidad promovida por el acceso a un archivo o parte de él desde cualquier nodo.
- Es altamente escalable y extensible.
- Aprovechan capacidad de procesamiento no utilizada de computadoras locales.

Desventajas:

- Muchos nodos pueden procesar la misma búsqueda.
- Excesiva carga de trabajo al replicar comunicaciones entre pares.
- Preocupaciones por la seguridad y la confianza.

Cuando se usa:

- Donde el sistema es de cómputo intensivo y es posible separar el procesamiento requerido en un gran número de cálculos independientes.
- Donde el sistema implica el intercambio de información entre computadoras individuales en una red y no hay necesidad de que esta información se almacene o gestione de manera centralizada.

Arquitectura *Map-Reduce*

- Apropiado para el procesamiento de grandes conjuntos de datos, tales como los motores de búsqueda de internet o las redes sociales.
- Conceptualmente, simples programas deben ejecutar poco a poco grandes conjuntos de datos como si se hubiera utilizado una sola computadora.
- Este estilo permite la computación extendida entre múltiples computadoras. Dado que el riesgo de falla incrementa con la cantidad de computadoras utilizadas, este estilo permite recuperación frente a fallas.
- Los grandes conjuntos de datos se dividen en conjuntos más pequeños y son almacenados en un sistema de ficheros global. Uno o más de estos conjuntos de datos son procesados por un *map worker* produciendo resultados intermedios almacenados en sistemas de archivos locales.
- Los desarrolladores solo necesitan razonar sobre la corrección de cómo cada máquina procesa un único fragmento de datos.
- Se realiza mucho procesamiento paralelo que es transparente para el desarrollador.
- Solo debe asegurarse que cada trabajador (map o reduce) implemente su función correctamente.
- Debe distribuirse los fragmentos equitativamente dado que la performance general del sistema se degrada si algún *map worker* se demora más que los otros.
- Un *reduce worker* lee los resultados de los ficheros locales, los combina para producir el resultado final y almacenarlo en el sistema de fichero global.

Calidad resultante:

- Escalabilidad: si el programa se escribió para este estilo puede correr en un clúster de una o mil máquinas.
- Disponibilidad: se recupera de fallas re-agendando el trabajo a otra máquina.
- Performance: tareas pueden distribuirse en varias máquinas.

Arquitecturas de la vista de distribución

1. Arquitectura Espejada (*Mirrored*).
2. Arquitectura *Rack*.
3. Arquitectura Granja de Servidores (*Farm*).

Arquitectura *Mirrored*

- En este estilo se duplica hardware idéntico, que corre en paralelo.
- Generalmente opera en “lock-step”, lo que duplica el esfuerzo, pero si uno falla el otro puede continuar por su cuenta. Realiza la misma operación varias veces en paralelo.
- El software puede actualizarse en forma separada uno de otro.
- La disponibilidad aumenta porque la probabilidad de que ambas fallen simultáneamente es baja.

Arquitectura *Rack*

- En este estilo los servidores se acomodan en pilas, para utilizar menos espacio del piso.
- Todas las computadoras de la pila se conectan con la misma red.
- La red en cambio puede tener varias conexiones a internet.
- La conexión a red para el rack es a menudo más rápida o la restricción de ancho de banda es menor, por lo que la comunicación entre computadoras de mismo rack es mucho más rápida.

Arquitectura granja de servidores (*Farm*)

- En este estilo muchas computadoras (usualmente idénticas) son ubicadas en la misma habitación (granja).
- La interconexión de las computadoras puede variar y la granja puede componerse de varios racks.
- Se las piensa como un recurso masivo que puede alojar cualquier aplicación.
- Es fácilmente escalable agregando más hardware del mismo tipo.
- Usos comunes para las granjas: capa de interfaz de usuario y capa intermedia de un sistema de tres capas donde cada granja aloja una capa diferente.

ARQUITECTURA ORIENTADA A SERVICIOS

Introducción

Basadas en la noción de servicio Web.

Al usar un servicio web, las organizaciones que quieran hacer accesible su información para otros programas pueden lograrlo al definir y publicar una interfaz de servicio web.

- **Servicio**: una acción ofrecida por una parte a otra. Aunque el proceso puede ser atado a un producto físico, la performance es esencialmente intangible y normalmente no resulta en posesión de alguno de los factores de producción.
- **Servicio web**: es un enfoque estándar para hacer un componente reusable, disponible y accesible a través de la web.

La particularidad de un servicio es que el hecho de proveer el servicio es independiente de la aplicación que usa el servicio.

Las arquitecturas orientadas a servicios (SOA, *Service Oriented Architecures*) son una forma de desarrollar sistemas distribuidos en la que los componentes del sistema son servicios independientes y se ejecutan en computadoras distribuidas geográficamente. Los servicios son independientes de la plataforma y del lenguaje de implementación. Los sistemas de software pueden componerse de servicios locales y servicios externos de diferentes proveedores con interacción uniforme con los servicios del sistema.

Estándares

Los servicios están basados en estándares XML, acordados tal que pueden ser provistos en cualquier plataforma y escritos en cualquier lenguaje de programación. Estándares claves:

- **SOAP (*Simple Objetc Access Protocol*)**: estándar de intercambio de mensajes que soporta la comunicación entre servicios. Es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML.
- **WSDL (*Web Service Definition Language*)**: estándar para la definición de interfaz de servicio. Establece como deben definirse las operaciones de servicios y los enlaces de servicio. WSDL describe la interfaz pública a los servicios Web (operaciones y parámetros). Está basado en XML y describe la forma de comunicación, es decir, los requisitos del protocolo y los formatos de los mensajes necesarios para interactuar con los servicios

listados en su catálogo. Las operaciones y mensajes que soporta se describen en abstracto y se ligan después al protocolo concreto de red y al formato del mensaje.

- **WW-BPEL**: estándar para un lenguaje de flujo de trabajo que se usa para definir programas de proceso que implican varios servicios diferentes.
- **UDDI (*Universal Description Discovery and Integration*)**: define los componentes de una especificación de servicio, que puede usarse para descubrir la existencia de un servicio. Desplazado por los motores de búsqueda.

Construir aplicaciones con base en servicios permite a las compañías y otras organizaciones cooperar y usar de manera mutua las funciones empresariales.

Ingeniería de servicio

La ingeniería de servicio es el proceso de desarrollo de servicios para reutilización en aplicaciones orientadas a servicios. Los ingenieros de servicios deben garantizar que el servicio represente una abstracción de reutilización que podría ser útil en diferentes sistemas. Deben documentar el servicio de modo que puedan descubrir y comprender los usuarios potenciales.

Existen tres etapas lógicas en el proceso de ingeniería de servicio:

1. **Identificación de candidatos a servicio:** se identifican los posibles servicios que se podrían implementar y se definen los requerimientos del servicio.
2. **Diseño del servicio:** se diseñan las interfaces lógica y de servicio WSDL.
3. **Implementación y despliegue del servicio:** se implementa el servicio, se prueba y se pone a disposición del usuario.

Identificación de candidatos a servicio

La identificación de candidatos a servicio, implica comprender y analizar los procesos empresariales de la organización para decidir cuáles servicios de reutilización podrían implementarse para soportar dichos procesos.

Hay tres tipos de servicios fundamentales que pueden identificarse:

- **Servicios utilitarios:** se trata de servicios que implementan alguna funcionalidad general que pueden usar diferentes procesos empresariales (por ejemplo la conversión de divisas).
- **Servicios empresariales:** se trata de servicios asociados con una función empresarial específica (por ejemplo la inscripción de un estudiante a un curso).
- **Servicios de coordinación o proceso:** se trata de servicios que soportan un proceso empresarial más general que por lo general implican diferentes actores y actividades (por ejemplo la gestión de pedidos, donde participan proveedores, bienes y pagos realizados.

Los servicios pueden considerarse orientados a tareas u orientados a entidades.

- **Servicios orientados a tareas:** servicios asociados con una actividad
- **Servicios orientados a entidades:** son como objetos. Se asocian a una entidad empresarial.

Los servicios utilitarios o empresariales pueden orientarse a entidades o a actividades, pero los servicios de coordinación siempre son orientados a tareas.

No hay fórmulas mágicas para decidir cuáles son los mejores servicios y, por lo tanto, la identificación de servicios es un proceso basado en habilidad y experiencia.

Diseño de interfaces de servicio

Esto implica definir operaciones asociadas con el servicio y sus parámetros. Su meta consiste en minimizar el número de intercambios de mensajes que deben tener lugar para completar la petición.

También hay que recordar que los servicios no tienen estado y que la gestión del estado de la aplicación específica del servicio es responsabilidad del usuario del servicio, y no del servicio en sí. Existen tres etapas en el diseño de interfaz de servicio:

- **Diseño de interfaz lógica:** identifican las operaciones asociadas al servicio, sus entradas y salidas y sus excepciones.
- **Diseño de mensajes:** se diseña la estructura de los mensajes que envía y recibe el servicio.
- **Desarrollo WSDL:** el diseño lógico y de mensajes se traducen a una descripción de interfaz abstracta escrita en WSDL.

Implementación y despliegue del servicio

Puede implicar la programación del servicio usando un lenguaje de programación estándar como Java o C#.

Una vez implementado un servicio, tiene que probarse antes de desplegarse. Se tratan de generar excepciones durante la prueba para comprobar que el servicio puede hacer frente a entradas inválidas.

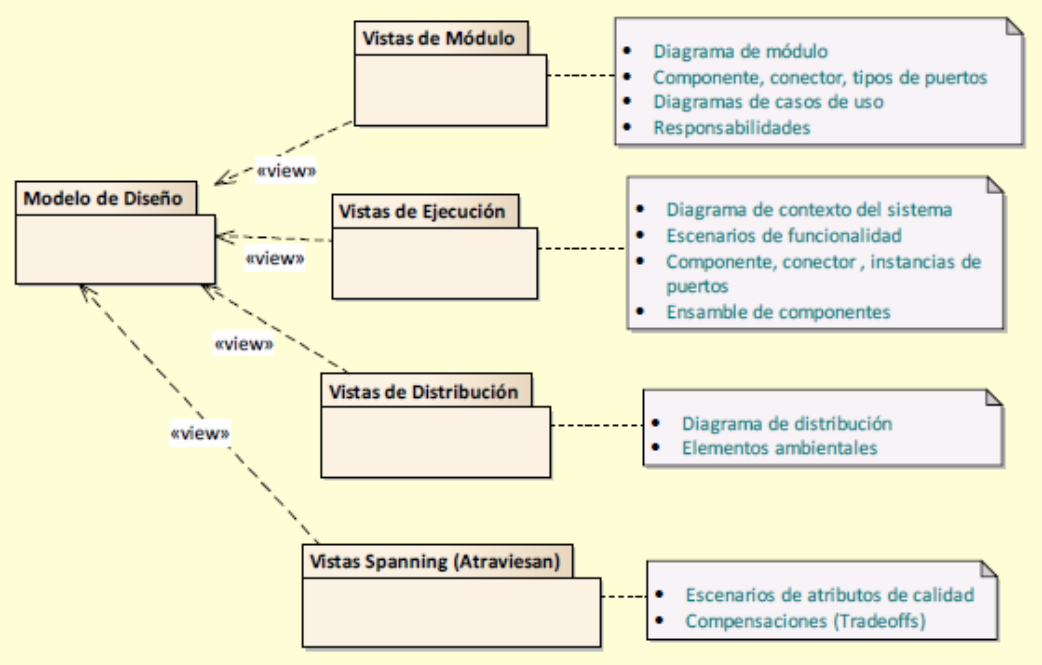
El despliegue del servicio, la etapa final del proceso, implica poner a disposición el servicio para su uso en un servidor Web.

Servicios de sistemas heredados

Los sistemas heredados son sistemas de software antiguos que emplea una organización. Dependen de tecnología obsoleta, pero todavía son esenciales para la empresa. Tal vez no sea efectivo en términos de costo reescribir o sustituir dichos sistemas, y muchas organizaciones quisieran usarlos en conjunción con sistemas más modernos.

Uno de los usos más importantes de los servicios es implementar envolturas (*wrappers*) para sistemas heredados que brinden acceso a las funciones y datos de un sistema. Entonces se puede acceder a dichos sistemas a través de la web e integrarlos con otras aplicaciones.

VISTAS ARQUITECTÓNICAS



Vista: las vistas son representaciones del sistema de acuerdo a una perspectiva. El interesado ve lo que necesita/quiere ver y se oculta la demás información. Las vistas se utilizan para representar la arquitectura con el uso de UML 2.0.

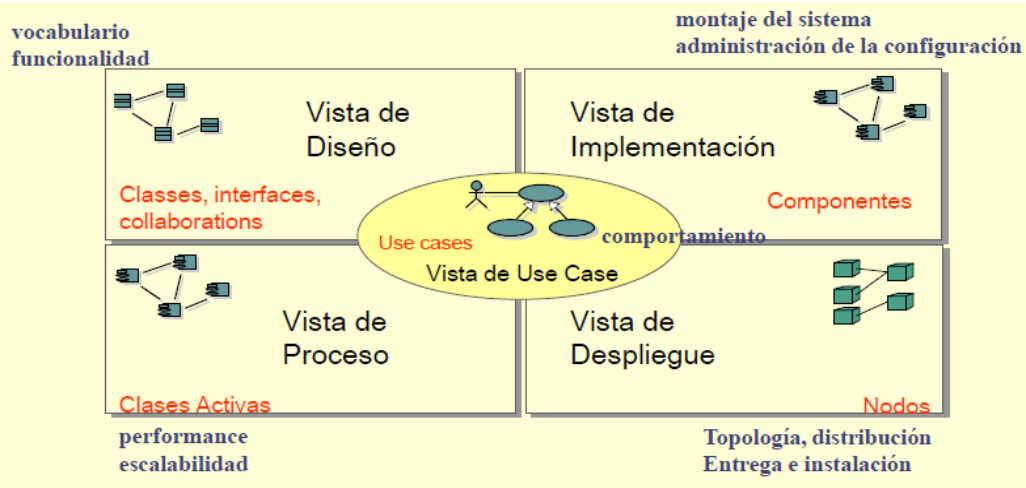
Tipos de vistas

- **Vista de módulo**: contiene vistas de los elementos que se pueden ver en tiempo de compilación. Vista estática de cómo organizamos el código. Definiciones de tipos de componentes, puertos y conectores, clases e interfaces.
    - Patrón Layered
- **Vista de ejecución**: contiene vistas de los elementos que se pueden ver en tiempo de ejecución. Incluye escenarios de funcionalidad, listas de responsabilidad y ensambles de componentes. Instancias de componentes, conectores y puertos.
    - Patrón Publish-Suscribe
    - Patrón Broker
    - Patrón Messaging
    - Patrón Process Coordinator
    - Patrón N-Tier
    - Patrón Pipe & Filter
    - Patrón Batch-Sequential
    - Patrón Modelo Repositorio
    - Patrón MVC
    - Patrón Peer-to-Peer
    - Patrón Map-Reduce
- **Vista de distribución**: contiene vistas de elementos relacionados con la distribución del software en el hardware.
    - Patrón N-Tier
    - Patrón Map-Reduce
    - Mirrored, Farm (granja) y Rack
- **Vista de spanning**: contiene cuestiones de confiabilidad, seguridad, performance, etc.

Los tipos componentes existen en el tipo de vista de módulo y las instancias de componentes (objetos) existen en la vista de ejecución, tal como ocurre con las clases y objetos.

Proveer vistas separadas ayuda a comprender cada asunto en forma aislada, pero también es necesario comprender la arquitectura como un todo.

Vistas en el Proceso Unificado de Desarrollo (PUD)



No todos los sistemas requieren todas las vistas. Por ejemplo:

- Un solo procesador: saltear la vista de despliegue.
- Procesos simples: saltear la vista de procesos.
- Programas muy pequeños: saltear la vista de implementación.

También se pueden agregar otras vistas como: vista de los datos, vista de seguridad.

REUTILIZACIÓN DE SOFTWARE

Introducción

La ingeniería de software basada en la reutilización es una estrategia en la que se engrana el proceso de desarrollo para reutilizar el software existente.

El movimiento hacia el desarrollo basado en la reutilización fue en respuesta a las demandas para reducir los costos de producción y mantenimiento del software, entregar los sistemas con mayor rapidez y aumentar la calidad del software.

La disponibilidad de software reutilizable aumentó drásticamente. Se dispone de una enorme base de código de reutilización a bajo costo. Esto puede ser en la forma de librerías de programas o aplicaciones completas. Los estándares, como los estándares de servicios web, han facilitado el desarrollo de servicios generales y su reutilización a través de varias aplicaciones.

Las unidades de software que se reutilizan pueden ser de tamaños sustancialmente diferentes:

- **Reutilización del sistema de aplicación**: todo un sistema de aplicación puede reutilizarse al incorporarlo sin cambios en otros sistemas o al configurar la aplicación para diferentes clientes.
- **Reutilización de componentes**: los componentes de una aplicación (desde subsistemas a objetos individuales) pueden reutilizarse.
- **Reutilización de objetos y funciones**: pueden reutilizarse los componentes de software que implementan una sola función, tal como una función matemática o una clase de objeto. Muchas librerías de funciones y clases están disponibles gratuitamente. Este enfoque es particularmente efectivo en áreas como algoritmos matemáticos y gráficas, donde se necesita experiencia especializada para desarrollar objetos y funciones eficientes.

Una forma complementaria de reutilización es la “reutilización de concepto” en la que, en vez de reutilizar un componente de software, se reutiliza una idea, una vía, un funcionamiento o un algoritmo. El concepto que se reutiliza se representa en una notación abstracta, que no incluye detalles de implementación. Por lo tanto, puede adaptarse y configurarse para varias situaciones.

Una ventaja evidente de la reutilización de software es que deberían reducirse los costos totales de desarrollo. No obstante, existen costos y problemas asociados con la reutilización. Hay un costo significativo asociado con entender si un componente es adecuado o no para su reutilización en una situación particular, y probar dicho componente para garantizar su confiabilidad.

En particular, debe haber una etapa de corrección de requerimientos en la que se modifiquen los requerimientos del sistema para reflejar el software de reutilización disponible.

La reutilización de software es más efectiva cuando se planea como parte de un programa de reutilización de toda la organización.

Ventajas de la reutilización

- **Confiabilidad creciente:** el software de reutilización, que se experimentó y ensayó en sistemas operativos, debe ser más confiable que el software nuevo.
- **Reducción de riesgo de proceso:** se conoce ya el costo del software existente, mientras que el de desarrollo siempre es una cuestión de juicio.
- **Uso efectivo de especialistas:** en lugar de hacer el mismo trabajo una y otra vez, los especialistas de aplicación pueden desarrollar software de reutilización que encapsule conocimientos.
- **Cumplimiento de estándares:** algunos estándares como los de interfaz de usuario, pueden implementarse como un conjunto de componentes de reutilización.
- **Desarrollo acelerado:** la reutilización de software puede acelerar la producción del sistema, ya que pueden reducirse los tiempos de desarrollo y validación.

Desventajas de la reutilización

- **Costos crecientes de mantenimiento:** si no está disponible el código fuente de un sistema o componente de software de reutilización, entonces los costos de mantenimiento podrían ser superiores, porque los elementos reutilizados del sistema pueden volverse cada vez más incompatibles con los cambios del sistema.
- **Falta de apoyo de herramientas:** algunas herramientas de software no apoyan el desarrollo con reutilización. Tal vez sea difícil o imposible integrar dichas herramientas con un sistema de librería de componentes.
- **Síndrome de "no se inventó aquí":** algunos ingenieros de software prefieren rescribir los componentes, porque consideran que pueden mejorarlos. Escribir software original se observa como más desafiante que reutilizar el software de alguien más.
- **Creación, mantenimiento y uso de una librería de componentes:** hay que adaptar procesos de desarrollo para asegurar que se use la librería.
- **Descubrimiento, comprensión y adaptación de componentes de reutilización:** deben descubrirse componentes de software en una librería, entenderse y, en ocasiones, adaptarse para trabajar en un nuevo entorno.

Panorama de la reutilización

Durante los últimos 20 años, se han desarrollado muchas técnicas para apoyar la reutilización de software. Dichas técnicas aprovechan el hecho de que los sistemas en el mismo dominio de aplicación son similares y tienen potencial de reutilización.

Los factores clave que deben considerarse al planear la reutilización son:

1. El calendario de desarrollo para el software: si el software debe desarrollarse rápidamente, se debe tratar de reutilizar sistemas comerciales en vez de componentes individuales.
2. La vida esperada del software: para un sistema de prolongada duración, hay que enfocarse en la capacidad de mantenimiento del sistema.
   Durante su vida útil, se podrá adaptar el sistema a nuevos requerimientos, lo que significará hacer cambios a partes del sistema. Si no se tiene acceso al código fuente, es preferible evitar los componentes COTS y los sistemas de proveedores externos.
3. Los antecedentes, las habilidades y la experiencia del equipo de desarrollo: todas las tecnologías de reutilización son bastante complejas, y es necesario mucho tiempo para entenderlas y usarlas de manera efectiva.
4. La criticidad del software y sus requerimientos no funcionales: para un sistema crítico que deba certificarse mediante un regulador externo, quizá se deba crear un caso de confiabilidad para el sistema. Eso es difícil si no se tiene acceso al código fuente del software.

5. El dominio de la aplicación: en algunos dominios de aplicación existen muchos productos genéricos que pueden reutilizarse al configurarlos para una situación local.
6. La plataforma en la que operará el sistema: sistemas genéricos de aplicación pueden ser especificados de plataforma y sólo podrán reutilizarse si el sistema está diseñado para la misma plataforma.

Frameworks de aplicación

Ahora se ha vuelto claro que la reutilización orientada a objetos tiene mejor apoyo en un proceso de desarrollo orientado a objetos a través de abstracciones de grano grueso, llamadas frameworks (estructuras).

Framework: un conjunto integrado de artefactos de software (tales como clases, objetos y componentes), que colaborarán en la facilitación de una arquitectura de reutilización para una familia de aplicaciones relacionadas.

Los frameworks brindan soporte para características genéricas que es probable que sean utilizadas en todas las aplicaciones de un tipo similar.

Los frameworks apoyan la reutilización de diseño en la que ofrecen una arquitectura que sirve de esqueleto para la aplicación, así como la reutilización de clases específicas en el sistema. La arquitectura se define por las clases de objetos y sus interacciones. Las clases se reutilizan directamente y pueden extenderse utilizando características como la herencia.

Los frameworks se implementan como una colección de clases de objetos concretos y abstractos en un lenguaje de programación orientado a objetos. Por lo tanto, los frameworks son específicos del lenguaje. Es posible usar un framework para crear una aplicación completa o implementar parte de una aplicación (como la GUI por ejemplo).

Clases de frameworks

Hay tres clases de frameworks:

- **Frameworks de infraestructura de sistema:** apoyan el desarrollo de infraestructura de sistema como comunicaciones, interfaces de usuario y compiladores.
- **Frameworks de integración de middleware:** consisten en un conjunto de estándares y clases de objetos asociados que soportan comunicación de componentes e intercambio de información (ejemplo EJB).
- **Frameworks de aplicación empresarial:** se ocupan de dominios de aplicación específicos, tales como los sistemas de telecomunicaciones o financieros.

Para extender un framework no es necesario cambiar el código de éste. En vez de ello, nosotros agregamos clases concretas que heredan operaciones de clases abstractas en el framework. Además, quizá sea necesario definir callbacks (comunicaciones de regreso). Los callbacks son métodos que se solicitan en respuesta a eventos reconocidos por el framework. A esto se lo denomina “Inversión de Control”. Los objetos framework, y no los objetos específicos de aplicación, son los responsables del control en el sistema.

Los frameworks son un enfoque efectivo para la reutilización, aunque costosos para introducirse en procesos de desarrollo de software. Se consideran inherentemente complejos, y aprender a usarlos es una actividad que tal vez dure muchos meses. Depurar las aplicaciones basadas en frameworks resulta complicado, porque tal vez no se comprenda cómo interactúan los métodos del framework.

Líneas de productos de software

Uno de los enfoques más efectivos para la reutilización es la creación de líneas de productos de software o familias de aplicación.

Una línea de productos de software es un conjunto de aplicaciones con una arquitectura común y componentes compartidos, con cada aplicación especializada para reflejar diferentes requerimientos. El sistema central se diseña para configurarse y adaptarse a las necesidades de los diferentes clientes del sistema.

Las líneas de producto de software surgen por lo general de aplicaciones existentes. Esto es, una organización desarrolla una aplicación y, luego, cuando se requiere un sistema similar, el código de ésta se reutiliza de manera informal en la nueva aplicación. Sin embargo, el cambio tiende a corromper la estructura de la aplicación, así que, a medida que se desarrollan más instancias nuevas, se vuelve cada vez más difícil crear una nueva versión. Por ello, se puede diseñar una línea de productos genéricos. Esto implica identificar la funcionalidad común en instancias de producto e incluirlas en una aplicación base, que después se usa para desarrollo futuro. Esta aplicación base se estructura deliberadamente para simplificar la reutilización y la reconfiguración.

Frameworks vs líneas de productos de software

Los frameworks y las líneas de productos de software tienen mucho en común, pero las principales diferencias entre estos enfoques son las siguientes:

- Los frameworks de aplicación se apoyan en características orientadas a objetos, como herencia y polimorfismo, para implementar extensiones al framework. El código framework no se modifica. Las líneas de producto de software no necesariamente se crean mediante un enfoque orientado a objetos. Los componentes de aplicación cambian, se borran o rescriben.
- Los frameworks de aplicación se enfocan principalmente en brindar apoyo técnico antes que dominio especifico. Una línea de productos de software por lo general incrusta información detallada de dominio y de plataforma.
- Las líneas de producto de software generalmente son aplicaciones de control para equipo. Los frameworks de aplicación con frecuencia están orientados al software, y pocas veces ofrecen soporte para interfaz de hardware.

Reutilización de productos COTS

Un producto COTS (*Comercial-Off-The-Shelf*) es un sistema de software que puede adaptarse a las necesidades de diferentes clientes sin cambiar el código fuente del sistema.

Los productos COTS se adaptan al usar mecanismos de configuración internos que permiten que la funcionalidad del sistema se adecue a necesidades específicas del cliente.

Otras características de configuración permiten al sistema aceptar plug-ins que extiendan la funcionalidad o comprueben las entradas del usuario para garantizar que son válidas.

Ventajas de los productos COTS

- Es posible la implementación más rápida de un sistema fiable.
- Es posible ver qué funcionalidad ofrece la aplicación, de manera que es más fácil juzgar si es probable que sea adecuada o no.
- Se evitan algunos riesgos de desarrollo al usar software existente.
- Las empresas pueden enfocarse en su actividad central sin tener que dedicar muchos recursos al desarrollo de sistemas TI.

Desventajas de los productos COTS

- Tienen que adaptarse los requerimientos para reflejar la funcionalidad y el modo de operación del producto COTS.
- El producto COTS puede basarse en suposiciones que sean casi imposibles de cambiar. Por lo tanto, el cliente debe adaptar su empresa para reflejar dichas suposiciones.
- Elegir el sistema COTS correcto para una empresa puede ser un proceso difícil, en especial porque muchos productos COTS no están debidamente documentados.
- Los proveedores de productos COTS controlan el soporte y la evolución del sistema. Pueden salir del negocio, perder el control o incluso hacer cambios que ocasionen dificultados a los clientes.

Tipos de reutilización de productos COTS

- **Sistemas de solución COTS:** consisten en una aplicación genérica de un solo proveedor que se configura de acuerdo con los requerimientos del cliente.
- **Sistemas integrados COTS:** implican la integración de dos o más sistemas COTS (quizá de diferentes proveedores) para crear un sistema de aplicación.

INGENIERÍA DE SOFTWARE BASADA EN COMPONENTES (ISBC)

Introducción

Para el software personalizado, la ingeniería de software basada en componentes es una forma efectiva orientada a la reutilización para desarrollar nuevos sistemas empresariales.

La ingeniería de software basada en componentes (ISBC, CBSE – *Component-Based-Software-Engineering*) surgió a finales de los 90 como un enfoque al desarrollo de sistemas de software basada en la reutilización de componentes de software. Su creación fue motivada por la frustración de los diseñadores al percatarse de que el desarrollo orientado a objetos no conducía a una reutilización extensa. Las clases de objetos individuales eran muy detalladas y específicas. Por consiguiente, era casi imposible vender o distribuir objetos como componentes de reutilización individuales.

Los componentes son abstracciones de alto nivel en comparación con los objetos y se definen mediante interfaces.

La ISBC es el proceso de definir, implementar e integrar o componer los componentes independientes e imprecisos en los sistemas.

El desarrollo de software basado en componentes permite reutilizar piezas de código preelaborado que permiten realizar diversas tareas, conllevando a diversos beneficios como las mejoras a la calidad, la reducción del ciclo de desarrollo y el mayor retorno sobre la inversión.

Fundamentos de la ISBC

1. Componentes independientes que se especifican por completo mediante sus interfaces. La implementación de un componente puede sustituirse por otra, sin cambiar otras partes del sistema.
2. Estándares de componentes que facilitan la integración de estos. Definen, a un nivel mínimo, cómo deben especificarse las interfaces de componentes y cómo se comunican estos últimos. Si los componentes se conforman a los estándares, entonces su ejecución es independiente de su lenguaje de programación.
3. Middleware que brinda soporte para integración de componentes. Para hacer que componentes independientes distribuidos trabajen en conjunto, es necesario soporte de middleware que maneje las comunicaciones de componentes. El middleware para soporte de componentes maneja eficientemente los conflictos de bajo nivel y permite enfocarse en problemas relacionados con la aplicación.
4. Un proceso de desarrollo que se engrana con la ingeniería de software basada en componentes. Se necesita de un proceso de desarrollo que permita la evolución de requerimientos, dependiendo de la funcionalidad de los componentes disponibles.

Principios de diseño en la ISBC

1. Los componentes son independientes, de manera que sus ejecuciones no interfieren entre sí.
2. Se ocultan los detalles de la implementación. La implementación de componentes pueden cambiar sin afectar el resto del sistema.
3. Los componentes se comunican a través de interfaces bien definidas, donde es posible sustituir un componente por otro.
4. Las infraestructuras de componentes ofrecen varios servicios estándar que pueden usarse en sistemas de aplicación. Esto recude la cantidad de código nuevo que debe desarrollarse.

Componentes como servicios

Para apoyar esta visión de componente, se han desarrollado muchos y diferentes protocolos y estándares, como el Enterprise JavaBeans (EJB) de Sun, COM y .NET de Microsoft, y CCM de CORBA.

En la práctica, era imposible que componentes desarrollados mediante estos diferentes enfoques funcionaran juntos.

En respuesta a estos problemas, se desarrolló la noción de componente como un servicio, y se propusieron estándares para apoyar la ingeniería de software orientada a servicios.

La diferencia más significativa entre un componente como servicio y la noción original de componente es que los servicios son entidades independientes externas a un programa que los usa. Cuando se construye un sistema orientado al servicio, se hace referencia al servicio externo en vez de incluir en nuestro sistema una copia de dicho servicio.

Por lo tanto, la ingeniería de software orientada al servicio es un tipo de ingeniería de software basada en componentes.

Problemas de la ISBC

- **Confiabilidad de los componentes:** los componentes suelen ser cajas negras y el código no está disponible.
- **Certificación de componentes:** se deberían certificar los componentes para asegurar su confiabilidad, pero ¿Quién pagaría su certificación?
- **Predicción de propiedades emergentes:** al ser los componentes opacos, predecir sus propiedades emergentes es difícil. El sistema podría tener propiedades no deseadas.
- **Equilibro de requerimientos:** equilibro entre requerimientos ideales y componentes disponibles se hace intuitivamente. Se necesita de un método más estructurado y sistemático que ayude a seleccionar y configurar componentes.

Componentes

Un componente es:

- una unidad de software independiente que puede organizarse con otros componentes para crear un sistema de software.
- un elemento de software que se conforma a un modelo de componentes estándar y puede desplegarse y componerse independientemente sin modificación, de acuerdo con un estándar de composición.
- una unidad de composición con interfaces especificadas contractualmente y sólo con dependencias de contexto explícitas. Un componente de software puede implementarse independientemente y está sujeto a composición por terceras partes.

Lo que tienen en común estas definiciones es que concuerdan en que los componentes son independientes, y los consideran la unidad fundamental de composición de un sistema.

Componente: *es una pieza de código preelaborado que encapsula alguna funcionalidad expuesta a través de interfaces estándar.*

Características de los componentes

- **Estandarizado:** debe ajustarse a un estándar, que define interfaces, metadatos, documentación, composición e implementación.
- **Independiente:** debe ser factible componerlo e implementarlo sin usar otros componentes.
- **Componible:** todas las interacciones externas deben darse mediante interfaces definidas públicamente. Debe permitir acceso a información sobre sí mismo.

- **Implementable:** debe ser autocontenido. Capaz de ejecutarse como una entidad independiente. El componente es binario y no tiene que compilarse antes de su implementación.
- **Documentado:** deben implementarse de forma completa. Debe especificarse su sintaxis y la semántica de todas las interfaces de componente. Así los usuarios pueden decidir si el componente satisface sus necesidades.

Una forma útil de pensar en un componente es como un proveedor de uno o más servicios. Cuando un sistema necesita un servicio, llama a un componente que brinde ese servicio sin importar donde se está ejecutando o su lenguaje de programación.

1. El componente es una unidad ejecutable independiente definida mediante sus interfaces. Para usarlo no necesita conocimiento alguno de su código fuente. Puede hacerse referencia a él como un servicio externo o incluirse directamente en un programa.
2. Los servicios ofrecidos por un componente se ponen a disposición mediante una interfaz, y todas las interacciones por dicha interfaz. La interfaz del componente se expresa en términos de operaciones parametrizadas y nunca se expone su estado interno.

Una diferencia entre un componente como servicio y un componente como elemento de un programa es que los servicios son entidades independientes por completo. No tienen una interfaz "requiere". Diferentes programas pueden usar dichos servicios sin necesidad de implementar soporte adicional requerido ara el servicio.

Modelos de componentes

Un modelo de componentes es una definición de estándares para implementación, documentación y despliegue de componentes. Se establecen con la finalidad de que los desarrolladores de componentes se aseguren de que éstos pueden interoperar.

Los modelos más importantes son el modelo de Web Services, el modelo Java Enterprise JavaBeans y el modelo .NET de Microsoft.

Los elementos de un modelo de componentes definen las interfaces de componentes, la información que necesita usar el componente en un programa y cómo debe implementarse un componente.

- **Interfaces: e**l modelo de componentes especifica cómo deben definirse las interfaces y los elementos, tales como los nombres de operación, los parámetros y las excepciones que deben incluirse en la definición de la interfaz.
- **Uso:** para que los componentes se distribuyan y se acceda a ellos de manera remota, deben tener un nombre único asociado.
  Los metadatos de un componente son datos acerca del componente en sí, tales como información acerca de sus interfaces y atributos. Los metadatos son importantes porque permiten a los usuarios del componente determinar qué servicios se proporcionan y requieren. También puede especificar como pueden personalizarse los componentes binarios para un entorno de implementación particular.
- **Implementación:** el modelo de componentes incluye una especificación de cómo deben empacarse los componente para su implementación como entidades ejecutables independientes.

Middleware

Para componentes que se implementen como unidades de programa y no como servicios externos, el modelo de componentes establece los servicios a ofrecer por parte del middleware que apoye los componentes de ejecución.

Los servicios brindados por una implementación de modelo de componentes se dividen en dos categorías:

- **Servicios de plataforma:** permiten a los componentes comunicarse e interoperar en un entorno distribuido.
- **Servicios de soporte:** son servicios comunes que probablemente requieran muchos componentes diversos.

El middleware implementa los servicios de componentes y ofrece interfaces a dichos servicios. Para usar los servicios ofrecidos por la infraestructura de un modelo de componentes, se puede considerar a los componentes como implementados en un "contenedor". Un contenedor es una implementación de los servicios de soporte más una definición de las interfaces que debe proporcionar un componente para integrarlo en el contenedor. Cuando se usan otros componentes, no acceden directamente a las interfaces del componente, en vez de eso, se accede a éstas a través de una interfaz de contenedor que recurre al código para acceder a la interfaz del componente embebido.

Procesos ISBC

Los procesos ISBC son procesos de software que brindan soporte a la ingeniería de software basada en componentes.

Existen dos tipos de procesos ISBC:

- **Desarrollo para reutilización:** se ocupa del desarrollo de componentes o servicios que se reutilizaran en otras aplicaciones.
- **Desarrollo con reutilización:** proceso para desarrollar nuevas aplicaciones usando los componentes y servicios existentes.

Dichos procesos tienen diferentes objetivos y, por consiguiente, incluyen distintas actividades. En el proceso de desarrollo para reutilización, el objetivo es producir uno o más componentes reutilizables. En el desarrollo con reutilización, no sabe cuáles están disponibles, así que se necesita descubrir dichos componentes y diseñar un sistema para utilizarlos de la manera más efectiva. No puede tener acceso al código fuente del componente.

1. Adquisición de componentes es el proceso de adquirir componentes para reutilización o desarrollo en un componente reutilizable.
2. La gestión de componentes se ocupa de catalogar los componentes y almacenarlos y organizarlos para su uso.
3. La certificación de componentes comprueba componentes para asegurar que cumplen su especificación.

ISBC para reutilización

La ISBC para reutilización es el proceso de desarrollar componentes reutilizables y ponerlos a disposición para reutilizarlos a través de un sistema de gestión de componentes.

Para elaborar componentes de reutilización, deben adaptarse y extenderse componentes específicos de aplicación para crear versiones más genéricas y, por lo tanto, más reutilizables.

Los cambios que se pueden hacer a un componente para volverlo más reutilizable, incluye:

- Eliminar métodos específicos de aplicación.
- Cambiar los nombres para hacerlos más generales.
- Agregar métodos para brindar cobertura funcional más completa.
- Hacer manejadores de excepción consistentes para todos los métodos.

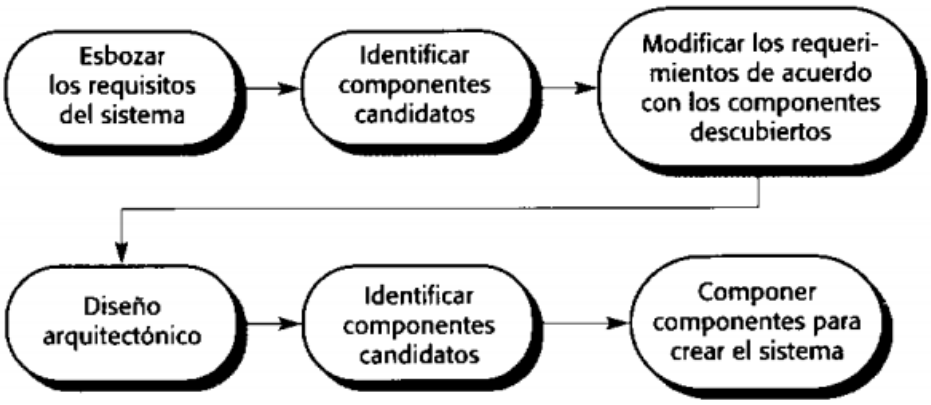
- Adicionar una interfaz de “configuración” para permitir la adaptación de los componentes a diferentes situaciones.
- Integrar los componentes requeridos para aumentar la independencia.

Conforme se agrega generalidad a un componente, aumenta la probabilidad de su reutilización. No obstante, esto por lo regular significa que el componente tiene más operaciones y más complejidades, las cuales lo hacen difícil de entender y de usar.

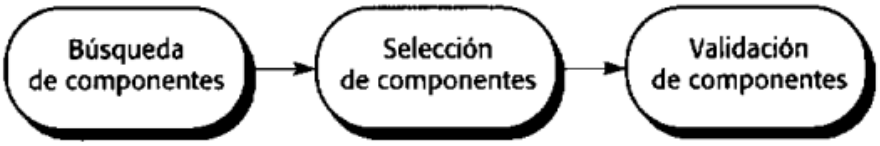
Hacer un componente reutilizable significa ofrecer una interfaz simple y mínima, que sea fácil de entender. En consecuencia, cuando se diseña un componente de reutilización, debe encontrar un compromiso entre la generalidad y comprensibilidad.

ISBC con reutilización

La ISBC con reutilización debe incluir actividades que encuentren e integren componentes reutilizables.



El proceso de identificación de componentes



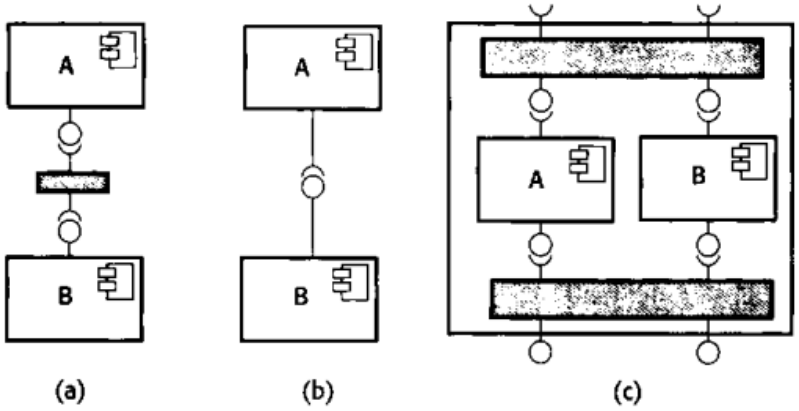
La validación de componentes consiste en probar los componentes para asegurarse de que se comportan como se espera. Consiste en desarrollar un conjunto de casos de prueba para un componente.

Composición de componentes

La composición de componentes es el proceso de integrar componentes uno con otro y con “código pegamento” especialmente escrito para crear un sistema u otro componente.

- **Composición secuencial (a):** se requiere código de pegamento adicional para llamar a los servicios del componente en el orden correcto y asegurar que los resultados entregados por el componente A sean compatibles con las entradas esperaras por el componente B. Este tipo de composición puede usarse con componentes que son elementos de programa o con componentes que son servicios.
- **Composición jerárquica (b):** ocurre cuando un componente llama directamente a los servicios que ofrece otro componente. El componente llamado proporciona los servicios que requiere el componente que llama. Este modo de composición no se usa cuando se implementan componentes como servicios web.

- **Composición aditiva (c):** ocurre cuando dos o más componentes se juntan (se suman) para crear un nuevo componente, lo que combina su funcionalidad. Este tipo de componente puede usarse con componentes que son unidades de programa o con componentes que son servicios.



Incompatibilidad de interfaces

- **Incompatibilidad de parámetros:** las operaciones de cada lado de la interfaz tiene el mismo nombre, pero sus tipos de parámetro o el número de parámetros son diferentes.
- **Incompatibilidad de operación:** los nombres de las operaciones en las interfaces "proporciona" y "requiere" son diferentes.
- **Operación incompleta:** la interfaz "proporciona" de un componente es un subconjunto de la interfaz "requiere" de otro componente o viceversa.

Beneficios

- Estudios informan que el ensamblaje de componentes lleva a :
  - Una reducción del 70% del tiempo de ciclo de desarrollo.
  - Un 84% del costo del proyecto.
  - Un índice de productividad del 26,2 comparado con la norma de industria del 19,9.
- Aunque estos resultados están en función de la robustez de la biblioteca de componentes, no hay duda que el ensamblaje de componentes proporciona ventajas significativas para los ingenieros de software.

Ventajas

- Aumento de reutilización de software.
- Simplifica las pruebas. Pruebas unitarias antes de probar el conjunto completo de componentes ensamblados.
- Simplifica el mantenimiento del sistema. Cuando existe un débil acoplamiento entre componentes, el desarrollador es libre de actualizar y/o agregar componentes según sea necesario, sin afectar otras partes del sistema.
- Mayor calidad. Dado que un componente puede ser construido y luego mejorado continuamente por un experto u organización, la calidad de una aplicación basada en componentes mejorará con el paso del tiempo.

ESTRATEGIAS DE PROTOTIPADO

Conceptos

Estrategia de prototipado

Una estrategia de prototipado es una elección de modelo de proceso (ciclo de vida) que se recomienda elegir a la hora de implementar un proyecto complejo, con dominio no familiar, que utilizará una tecnología desconocida.

Por ello se requiere el uso de prototipos en el diseño y la implementación, además de utilizarlos durante la validación de requerimientos.

Prototipo

Un prototipo consiste en una primera versión de un nuevo tipo de producto, en el que se han incorporado sólo algunas características del sistema final, o no se han realizado completamente.

Es un modelo o maqueta del sistema que se construye para comprender mejor el problema y sus posibles soluciones: evaluar mejor los requerimientos y probar opciones de diseño.

Un prototipo se puede utilizar de diferentes maneras en el proceso de desarrollo:

- En el proceso de ingeniería de requerimientos: ayuda a obtener y validar los requerimientos del sistema.
- En el proceso de diseño del sistema: ayuda a explorar soluciones de software particulares y para apoyar a diseño de interfaces de usuario.
- En el proceso de pruebas: se puede utilizar para ejecutar pruebas *back-to-back* con el sistema que se entregará al cliente (se envían las mismas pruebas al sistema y al prototipo):

Permiten a los usuarios ver como el sistema apoya a su trabajo. Pueden adquirir nuevas ideas para los requerimientos y encontrar áreas fuertes y débiles en el software. Puede revelar errores y omisiones en los requerimientos propuestos.

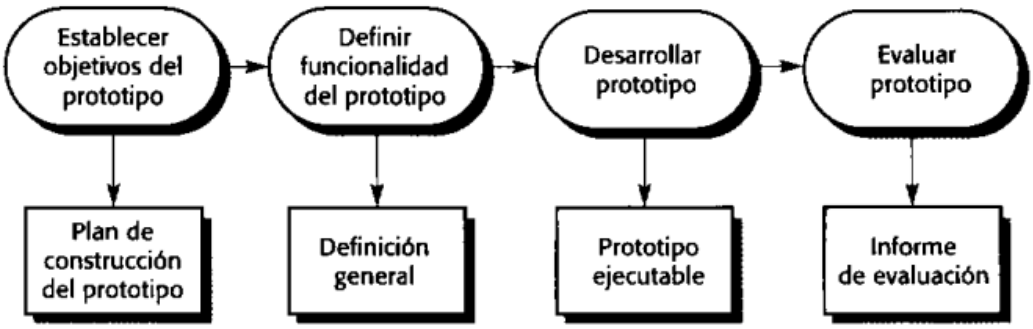
Características

- Funcionalidad limitada.
- Poca fiabilidad.
- Características de operación pobres.
- Pocos días de desarrollo.

Beneficios de los prototipos

- Mejora la usabilidad del sistema.
- Una mejor concordancia entre el sistema y las necesidades del usuario.
- Ayuda al cliente a establecer claramente los requerimientos.
- Mejora la calidad del diseño.
- Mejora el mantenimiento.
- Reducción en el esfuerzo de desarrollo.
- Aumentan la productividad.
- Planifican el desarrollo.
- Entusiasmo de los usuarios respecto a los prototipos.

Proceso de desarrollo de prototipos



1. **Establecer objetivos del prototipo:** los objetivos del prototipo deben ser claros desde el principio, ya que se puede malinterpretar la función del prototipo y no se obtengan los beneficios esperados.
2. **Definir funcionalidad del prototipo:** decidir que incluir y que excluir del prototipo. Se podría dejar a un lado los requerimientos no funcionales y centrarse en la funcionalidad, reducir estándares, obviar manejo de errores, etc. aunque depende del tipo de prototipo que se esté haciendo.
3. **Desarrollar prototipo:** construir el prototipo.
4. **Evaluar prototipo:** analizar los resultados del uso del prototipo.

¿Cuándo son interesantes los prototipos?

¡SIEMPRE!, pero especialmente cuando:

- Dominio riesgoso (bien por dificultad o por falta de tradición en su aplicación).
- Se usan nuevos métodos, técnicas, tecnologías.
- Es necesario evaluar previamente el impacto del sistema en los usuarios y en la organización.
- El costo de rechazo por parte de los usuarios, por no cumplir sus expectativas, es muy alto.

Clasificación de prototipos

Prototipos según su propósito

- **Prototipado de interfaz de usuario:** modelos de pantallas.
- **Prototipado funcional (operacional):** implementa algunas funciones, y a medida que se comprueba que son las apropiadas, se corrigen, refinan y se añaden otras.
- **Prototipos arquitectónicos:** sirven para evaluar decisiones arquitectónicas de infraestructura, tecnología e integración.
- **Modelos de rendimiento:** evalúan el rendimiento de una aplicación crítica (no sirven al análisis de requerimientos).

Prototipos según su utilidad

- **Prototipos rápidos o desechables**
  - Sirve al análisis y a la validación de requerimientos.
  - Después se redacta la especificación del sistema y se desecha el prototipo.
  - La aplicación se desarrolla siguiendo un paradigma diferente.
  - Problema: cuando el prototipo no se desecha y termina convirtiéndose en el sistema final.
- **Prototipos evolutivos**
  - Comienza con un sistema relativamente simple que implementa los requerimientos más importantes o mejor conocidos.
  - El prototipo se aumenta o cambia en cuanto se descubren nuevos requerimientos.
  - Finalmente, se convierte en el sistema requerido.
  - Se usa en el desarrollo Web y en aplicaciones de comercio electrónico.

Prototipos según su alcance

- **Vertical:** desarrolla completamente alguna de las funciones.
- **Horizontal:** desarrolla parcialmente todas las funciones.

DISEÑO DE INTERFACES DE USUARIO

Introducción

Un buen diseño de la interfaz de usuario es crítico para la confiabilidad del sistema. Muchos de los llamados "errores de usuario" son causados por el hecho de que las interfaces de usuario no consideran las habilidades de los usuarios reales y su entorno de trabajo. Una interfaz de usuario mal diseñada significa que los usuarios probablemente no podrán acceder a algunas características del sistema, cometerán errores y sentirán que el sistema les dificulta en vez de ayudarlos a conseguir cualquier objetivo para el que utilizan el sistema.

Factores humanos en el diseño de GUIs

1. Las personas tienen una memoria limitada a corto plazo: los humanos podemos recordar hasta siete elementos de información. Si a los usuarios se les presenta demasiada información al mismo tiempo, es posible que no puedan asimilarla.
2. Las personas cometen errores: especialmente cuando manejamos mucha información o estamos estresados. Cuando los sistemas fallan y emiten mensajes de aviso y alarmas, a menudo aumentan el estrés de otros usuarios, incrementando así la posibilidad de cometer errores.
3. Las personas son diferentes: muchas personas tienen diferentes capacidades físicas. No se debe diseñar para sus propias cualidades físicas y suponer que todos los usuarios serán capaces de adaptarse.
4. Las personas tienen diferentes preferencias de interacción: a algunas personas les gustan las imágenes mientras que a otras el texto.

Principios de diseño de GUIs

1. **Familiaridad del usuario:** la interfaz debe utilizar términos familiares para los usuarios, y los objetos que el sistema manipula deben estar directamente relacionados con el entorno de trabajo del usuario.
2. **Uniformidad:** significa que los comandos y menús del sistema deben tener el mismo formato, los parámetros deben pasarse a todos los comandos de la misma forma, y la puntuación de los comandos debe ser similar. Esto reduce el tiempo de aprendizaje del usuario.
3. **Mínima sorpresa:** es apropiado debido a que las personas se irritan demasiado cuando el sistema se comporta de forma inesperada. Los usuarios no deberían sorprenderse por el comportamiento del sistema.
4. **Recuperabilidad:** los usuarios inevitablemente cometen errores cuando utilizan un sistema. Por eso, se deben incluir recursos que permitan a los usuarios recuperarse de sus errores:
    a. Confirmación de acciones destructivas: cuando un usuario va a realizar una acción crítica o potencialmente destructiva, el sistema deberá pedirle que confirme que esto es realmente lo que desea antes de destruir cualquier información.
    b. Proporcionar un recurso para deshacer: restablece el sistema al estado previo antes de que ocurriera la acción.
    c. Generar puntos de control: implica grabar el estado de un sistema en intervalos periódicos y permitir que el sistema se restaure desde el último punto de control.
5. **Asistencia al usuario:** las interfaces deben proporcionar asistencia al usuario o características de ayuda. Estas se deben integrar en el sistema y proporcionar diferentes niveles de ayuda y asesoramiento.
6. **Diversidad de usuarios:** para un sistema pueden existir diferentes tipos de usuarios. Por ejemplo, usuarios casuales usaran asistentes, y usuarios potenciales usaran métodos abreviados de forma que puedan interactuar con el sistema lo más rápido posible.

Interacción de usuario

Cinco formas de interacción:

1. **Manipulación directa:** el usuario interactúa directamente con los objetos de la pantalla.
    a. Ventajas
        i. Interacción rápida e intuitiva
        ii. Fácil de aprender
    b. Desventajas
        i. Puede ser difícil de implementar
        ii. Adecuado únicamente cuando hay una metáfora visual para tareas y objetos
    c. Ejemplo
        i. Videojuegos y sistemas CAD
2. **Selección de menús:** el usuario selecciona un comando de una lista de posibilidades.
    a. Ventajas
        i. Evita errores del usuario
        ii. Se requiere poco tipeo
    b. Desventajas
        i. Lento para usuarios experimentados
        ii. Puede ser complejo si hay muchas opciones
    c. Ejemplo
        i. La mayoría de sistemas de propósito general
3. **Llenado de formularios:** el usuario rellena los campos de un formulario.
    a. Ventajas
        i. Ingreso de datos simple
    b. Desventajas
        i. Ocupa mucho espacio en la pantalla
        ii. Causa problemas cuando la opción del usuario no coincide con los campos del formulario
    c. Ejemplo
        i. Control de stock, sistemas generales, etc.
4. **Lenguaje de comandos:** el usuario emite un comando especial y los parámetros asociados para indicar al sistema que hacer. Es la típica consola.
    a. Ventajas
        i. Poderoso y flexible
    b. Desventajas
        i. Difícil de aprender
        ii. Manejo de errores pobre
    c. Ejemplo
        i. Sistemas operativos, sistemas de control de comandos
5. **Lenguaje natural:** el usuario emite un comando en lenguaje natural.
    a. Ventajas
        i. Accesible a usuarios casuales
        ii. Fácilmente extensible
    b. Desventajas
        i. Requiere más tipeo.
    c. Ejemplo
        i. Buscadores

Estos estilos de interacción se pueden mezclar, utilizando varios estilos en la misma aplicación.

Presentación de la información

Todos los sistemas interactivos tienen que proporcionar alguna forma de presentar la información a los usuarios. La presentación de la información puede ser simplemente una representación directa de la información de entrada (texto) o presentar información gráficamente. Una buena pauta de diseño es mantener separado el software requerido para la presentación de la información misma. Separar el sistema de presentación de los datos nos permite cambia la representación en la pantalla del usuario sin tener que cambiar el sistema de cálculo subyacente.

El enfoque MVC es una forma efectiva para permitir representaciones múltiples de datos. Por lo tanto, un modelo que representa datos numéricos puede tener una vista que represente los datos como un histograma y una vista que presente los datos como una tabla.

Cuando se decide como presentar la información, deben tenerse en cuenta las siguientes cuestiones:

- ¿El usuario está interesado en información precisa o en las relaciones entre los valores de los datos?
- ¿Con que frecuencia cambian los valores de la información?
- ¿El usuario debe tomar alguna acción en respuesta a un cambio de información?
- ¿Existe una interfaz de manipulación directa?
- ¿La información a mostrar es textual o numérica?

Gráficos vs texto

Los gráficos muestran la información más directamente, pero ocupan un valioso espacio en la pantalla y pueden tardar bastante tiempo en descargarse si el usuario está trabajando con una conexión lenta.

La precisión textual ocupa menos espacio en la pantalla, pero no puede ser vista de un vistazo.

- **Texto:** se debe usar cuando la información tiene que ser precisa y cambie de forma relativamente lenta.
- **Gráfico:** se debe usar cuando los datos cambian rápidamente o si las relaciones entre los datos son más importantes que los valores exactos.

Las vistas digitales que cambian constantemente pueden ser confusas y molestas ya que los lectores no pueden leer y asimilar la información antes de que cambie. Por lo tanto, la información numérica que varía dinámicamente se representa mejor gráficamente utilizando una representación analógica. Si es necesario, las vistas gráficas pueden complementarse con una vista digital precisa.

Tipos de información

- **Información estática:** inicializada al principio de la sesión. No cambia durante la sesión. Puede ser numérica o textual.
- **Información dinámica:** cambiar durante la sesión y los cambios deben ser comunicados al usuario del sistema. Puede ser numérica o textual.

Tipos de presentación

- **Presentación digital:** es compacta. Necesita poco espacio en pantalla. Se utiliza para comunicar valores precisos.
- **Presentación análoga:** más fácil para obtener una impresión a primera vista de un valor. Es posible mostrar valores relativos. Más fácil de ver valores excepcionales de los datos. Ejemplo: gráficos, reloj analógico, termómetro, etc.

Colores

También, se debe pensar detenidamente en los colores utilizados en la interfaz. El color puede mejorar las interfaces de usuario ayudando a los usuarios a comprender y manejar la complejidad. Sin embargo, es fácil utilizar el color de forma errónea para crear interfaces visualmente poco atractivas y propensa a errores.

A tener en cuenta:

1. **Limitar el número de colores utilizados y ser conservador en la forma de utilizarlos:** no deben utilizarse más de cuatro o cinco colores diferentes en una ventana y no más de siete en una interfaz de sistema.
2. **Utilizar un cambio de color para mostrar un cambio en el estado del sistema:** si una vista cambia de color, debe significar que ha ocurrido un evento importante.
3. **Utilizar el código de colores para apoyar la tarea que los usuarios están tratando de llevar a cabo:** si los usuarios tienen que identificar instancias anómalas, se deben resaltar estas instancias; si también tiene que descubrir similitudes, se deben resaltar éstas utilizando un color diferente.
4. **Utilizar el código de colores de una forma consistente y uniforme:** si una parte de un sistema muestra los mensajes de error en rojo, todas las demás partes deben mostrarlos de igual forma. El rojo no se debe utilizar para nada más. Si se hace, es posible que el usuario interprete la vista en rojo como un mensaje de error.
5. **Ser cuidadoso al utilizar pares de colores:** hay combinaciones de colores que pueden ser visualmente molestas o difíciles de leer.

En general, se debe utilizar el color para la acción a resaltar, pero no se deben asociar significados con colores particulares.

Mensajes de error

Los sistemas se comunican con los usuarios a través de mensajes que proporcionan información sobre los errores y estados del sistema.

Factores que deben tenerse en cuenta a la hora de diseñar mensajes del sistema:

1. **Contexto:** los mensajes generados por el sistema deben reflejar el contexto actual del usuario. El sistema debe ser consciente de lo que está haciendo el usuario y generar mensajes relacionados con su actividad actual.
2. **Experiencia:** debería proveerse mensajes para usuarios experimentados y para usuarios principiantes, y permitirle al usuario que elija que tan conciso es el tipo de mensaje que desea.
3. **Nivel de habilidad:** los mensajes se deben adaptar a las habilidades del usuario. Los mensajes para diferentes clases de usuarios deberían presentar diferentes terminologías dependiendo de lo que es familiar para el usuario.
4. **Estilo:** los mensajes deberían ser positivos en lugar de negativos. Deberían estar escritos en modo activo y no pasivo. No deben ser insultantes o tratar de ser graciosos.
5. **Cultura:** cada vez que sea posible, el diseñador de mensajes debe estar familiarizado con la cultura del país donde se utilizará el sistema. Un mensaje adecuado para una cultura puede ser inadecuado para otras.

Los mensajes de error siempre deben ser formales, concisos, uniformes y constructivos. No deben ser ofensivos ni tener sonidos asociados u otro tipo de ruidos que pueden desconcertar al usuario. En la media de lo posible, el mensaje debe sugerir cómo se podría corregir el error. El mensaje de error debe vincularse a un sistema de ayuda en línea sensible al contexto.

El proceso de diseño de interfaces de usuario

El diseño de la GUI es un proceso iterativo donde los usuarios interactúan con los diseñadores y prototipos de la interfaz para decidir las características, organización, apariencia y funcionamiento de la interfaz de usuario del sistema.

Antes de que empiece la programación, debe haber desarrollado, e idealmente, probado algunos diseño en papel.

Existen tres actividades en este proceso:

1. **Análisis del usuario:** se desarrolla una comprensión de las tareas que éste realiza, su entorno de trabajo, los otros sistemas que utiliza, cómo interactúan con el resto de las personas en su trabajo, etc.
2. **Prototipado del sistema:** se deben desarrollar prototipos del sistema y exponerlos a los usuarios, quienes pueden entonces guiar la evolución de la interfaz.
3. **Evaluación de la interfaz:** actividad de evaluación donde se recopila información sobre las experiencias reales de los usuarios con la interfaz.

Análisis del usuario

Si no se entiende lo que los usuarios quieren hacer con el sistema, no se podrá llevar a cabo un diseño real y eficaz de la interfaz de usuario. Para desarrollar esta comprensión, se pueden utilizar técnicas como el análisis de tareas, estudios etnográficos, entrevistas de usuarios, cuestionarios y observaciones, o una mezcla de todas.

Prototipado de la interfaz de usuario

El prototipado evolutivo o exploratorio con la implicación de los usuarios finales es la única forma práctica de diseñar y desarrollar interfaces gráficas de usuario para sistemas software. Implicar al usuario en el proceso de diseño y desarrollo es un aspecto fundamental del "diseño centrado en el usuario".

El propósito del prototipado es permitir a los usuarios adquirir una experiencia directa con la interfaz.

Cuando se está construyendo el prototipo de una interfaz de usuario se debe adoptar un proceso de prototipado en dos etapas:

1. Al principio del proceso, hay que desarrollar prototipos de papel y mostrárselos a los usuarios finales.
2. Entonces, se perfecciona el diseño y se desarrollan prototipos automatizados cada vez más sofisticados, y se ponen a disposición de los usuarios para realizar pruebas y simulación de actividades.

La construcción de prototipos en papel es un proceso poco costoso y sorprendentemente efectivo para el desarrollo de prototipos.

Como alternativa, se puede utilizar una técnica basada en *storyboards* para presentar el diseño de interfaz. Un *storyboard* es una serie de esbozos que ilustran una secuencia de interacciones.

Enfoques de prototipado

- **Enfoque dirigido por secuencias de comandos:** se crean pantallas con elementos visuales, como botones y menús, y se asocia una secuencia de comandos con estos elementos.
- **Lenguajes de programación visuales:** lenguajes como Visual Basic, incorporan un potente entorno de desarrollo de GUIs que permite crear interfaces rápidamente.
- **Prototipado basado en internet:** basadas en navegadores web (HTML).

Evaluación de la interfaz

La evaluación de la interfaz es el proceso de evaluar la forma en que se utiliza una interfaz y verificar que cumple con los requerimientos del usuario.

Una evaluación se debe llevar a cabo contra una especificación de la usabilidad basada en atributos de la usabilidad.

Atributos:

- Aprendizaje: ¿Cuánto tiempo tarda un usuario nuevo en ser productivo con el sistema?
- Velocidad de funcionamiento: ¿Cómo responde el sistema a las operaciones de trabajo del usuario?
- Robustez: ¿Qué tolerancia tienen el sistema a los errores del usuario?
- Recuperación: ¿Cómo se recupera el sistema a los errores del usuario?

- Adaptación: ¿Está muy atado el sistema a un único modelo de trabajo?

La evaluación sistemática del diseño de la interfaz puede ser muy costosa. Existen varias técnicas menos costosas y sencillas en la evaluación de interfaces que pueden identificar deficiencias específicas en el diseño de interfaces:

- Cuestionarios que recopilan información de lo que opinan los usuarios de la interfaz.
- La observación de los usuarios cuando trabajan con el sistema y "piensan en voz alta" de cómo tratan de utilizar el sistema para llevar a cabo alguna tarea.

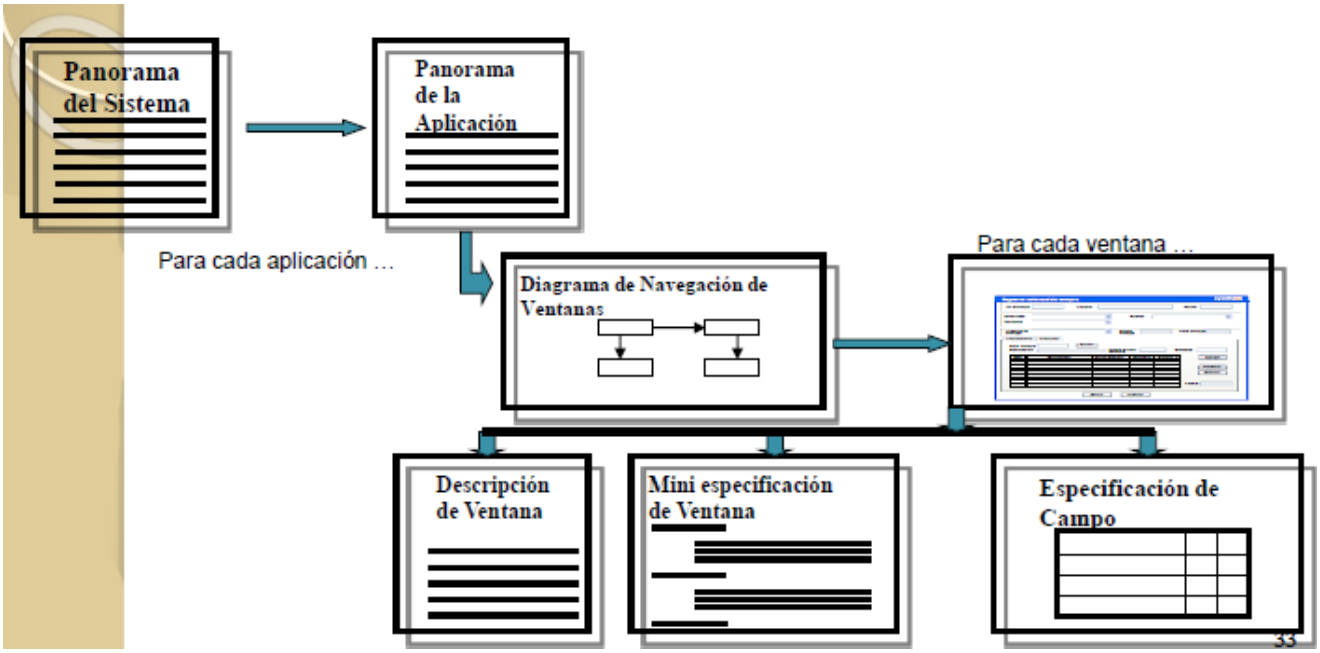
Consideraciones

- **Interacción general**
  - Ser consistente
  - Ofrecer respuestas significativas
  - Pedir verificación de cualquier acción destructiva
  - Permitir volver atrás en la mayoría de las acciones
  - Reducir la cantidad de información que se debe memorizar entre acciones
  - Perdonar los errores
  - Buscar la eficiencia en el diálogo, el movimiento y el pensamiento
  - Categorizar las actividades por función y organizar la pantalla de acuerdo a esto
  - Proporcionar ayudas sensibles al contexto
  - Usar verbos de acción sencillos o frases verbales cortas para nombrar las órdenes
- **Visualización de la información**
  - Mostrar información que sea relevante en el contexto actual
  - No abrumar al usuario con datos. Permitirle una rápida asimilación de la información
  - Usar etiquetas consistentes, abreviaciones estándar y colores predecibles
  - Permitir al usuario mantener el contexto visual (scroll, wizard)
  - Producir mensajes de error significativos
  - Usar mayúsculas y minúsculas, tabulaciones y agrupamiento de texto
  - Usar ventanas para compartimentar diferentes tipos de información
  - Usar presentaciones analógicas para representar información que es más fácilmente asimilable
  - Estudiar la distribución disponible en la pantalla y usarla eficientemente
- **Entrada de datos**
  - Minimizar el número de entrada de datos que necesita realizar el usuario
  - Mantener la consistencia entre la visualización y la introducción de datos
  - Permitir al usuario personalizar la entrada
  - La interacción debería ser flexible pero también sintonizarse al modo preferido del usuario de entrada
  - Desactivar las órdenes que son inapropiadas en el contexto de las acciones actuales
  - Dejar al usuario controlar el flujo interactivo
  - Proporcionar ayuda en todas las acciones de entrada
  - Eliminar entradas innecesarias

Las ocho reglas de Shneiderman

1. Buscar siempre la coherencia.
2. Permitir el uso de *shortcuts.*
3. Dar realimentación de información.
4. Diseñar diálogos que tengan un fin.
5. Permitir los manejos simples de los errores.
6. Permitir deshacer acciones con facilidad.
7. Permitir que el centro de control sea interno.
8. Reducir la carga de la memoria intermedia.

Productos del diseño de interfaz externa



DISEÑO ORIENTADO A OBJETOS

Características de un buen diseño

- Independencia de los componentes.
- Identificación y tratamiento de excepciones.
- Prevención y tolerancia de defectos.

Independencia de componentes

En la mayoría de los diseño se trata que los componentes sean independientes unos de otros. Para medir el grado de independencia de los componentes de un diseño se aplican dos conceptos: Acoplamiento y Cohesión.

Acoplamiento

- Dos componentes están altamente acoplados cuando existe mucha dependencia entre ellos.
- Los componentes poco acoplados, tienen algunas dependencias, pero las interconexiones entre ellos son débiles.
- Los componentes no acoplados no tienen interconexiones con el resto, son completamente independientes. Poco probable en un sistema.

Cohesión

- Se refiere al grado de interconexión interna con el que se ha construido el componente.
- Un componente es cohesivo si todos sus elementos están orientados a la realización de una única tarea y son esenciales para llevarla a cabo.

Principios de diseño

Un principio de diseño es una técnica o herramienta básica que puede ser aplicada para diseñar o construir software, para hacer más flexible, extensible y mantenible.

Principios de diseño básicos

- **Flexibilidad:** el software puede cambiar y crecer sin un constante re-trabajo. Evita la fragilidad del software.
- **Encapsulamiento:** mantiene las partes del software que son estables, lejos de las partes que cambian, entonces es fácil hacer cambios sin necesidad de romper todo.

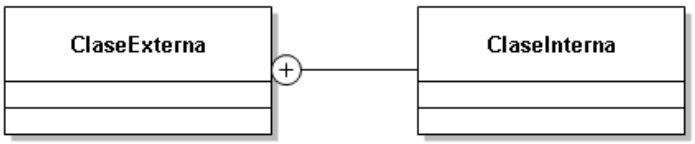
- **Funcionalidad:** sin esto no importa que bien esté hecho el diseño, el cliente no estará feliz.
- **Uso de patrones de diseño:** se trata de reúso y de no tratar de resolver un problema que ya resolvió alguien más.

Principios

Nº	Principio	Descripción
1	**Abierto-Cerrado**	• Es para permitir el cambio, y para permitirlo sin modificar lo existente. • Clases abiertas para extensión. • Clases cerradas para modificación. • Es a cerca de la flexibilidad.
2	**No te repitas (DRY)**	• Evitar duplicaciones, abstrayendo cosas comunes y ubicándolas en un único lugar. • Se trata de ubicar un requerimiento en un único lugar. • Tener cada pieza de información y comportamiento en un único lugar.
3	**Responsabilidad única**	• Cada objeto del sistema debería tener una única responsabilidad. • Todos los servicios del objeto deberían focalizarse en ejecutar esa única responsabilidad. • Se cumple si cada objeto tiene una única razón para cambiar (cohesión).
4	**Sustitución de Liskov**	• Los subtipos deben ser utilizados por sus tipos base. • Se refiere a la herencia bien diseñada. • Este principio revela problemas con la estructura de la herencia, si existieran.

Clases anidadas

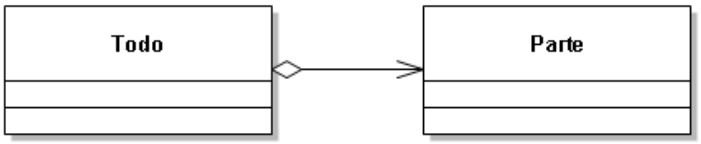
Algunos lenguajes como Java permiten situar una definición de clase dentro de otra definición de clase. Esto crea lo que se conoce como una clase anidada (*nested class*). Una clase anidada se declara dentro del espacio de nombres de su clase externa y es sólo accesible por esa clase o por los objetos de esa clase. En Java se conoce como clase interna (*inner class*).



Agregación

Es un tipo de relación todo-parte en la que el conjunto se compone de muchas partes. Es una relación todo-parte, un objeto (el todo) utiliza los servicios de otro objeto (la parte). La parte ni siquiera sabe que es parte de un todo.

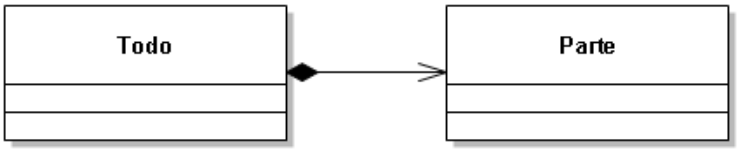
La agregación es asimétrica. Esto significa que un objeto nunca puede ser, directa o indirectamente, parte de sí mismo.



Composición

La composición es una forma más fuerte de agregación y tiene semántica similar. Al igual que la agregación, es una relación todo-parte. La diferencia clave entre agregación y composición es que en composición las partes no tienen vida independiente fuera del todo. En la composición, cada parte pertenece al menos a un y sólo a un todo, mientras que en agregación una parte se puede compartir entre los todos.

- Las partes pueden solamente pertenecer a un conjunto cada vez.
- El conjunto tiene responsabilidad única para la deposición de todas sus partes (creación y destrucción).
- El conjunto puede liberar partes, siempre y cuando la responsabilidad para ellas se asuma por otro objeto.
- Si se destruye el conjunto, debe destruirse todas sus partes.



La forma de reconocer a la agregación y a la composición es hacer la siguiente pregunta: "¿El objeto cuyo comportamiento quiero usar, existe fuera del objeto que usa ese comportamiento?". Si el objeto tiene sentido de existencia por si solo se debe usar agregación, de lo contrario usar composición.

Encapsulamiento

Los programas orientados a objetos están formados de objetos ☺. Un **objeto** encapsula tanto datos como los procedimientos que operan sobre estos datos (**métodos** u **operaciones**). Un objeto realiza una operación cuando recibe una **petición** (**mensaje**) de un **cliente**.

Las peticiones son el único modo de lograr que un objeto ejecute una operación. Las operaciones son la única forma de cambiar los datos internos de un objeto. Debido a estas restricciones, se dice que el estado interno de un objeto está **encapsulado**; no puede accederse a él directamente, y su representación no es visible desde el exterior del objeto.

Herencia

En la etapa de diseño la herencia se considera mucho más que en el análisis.

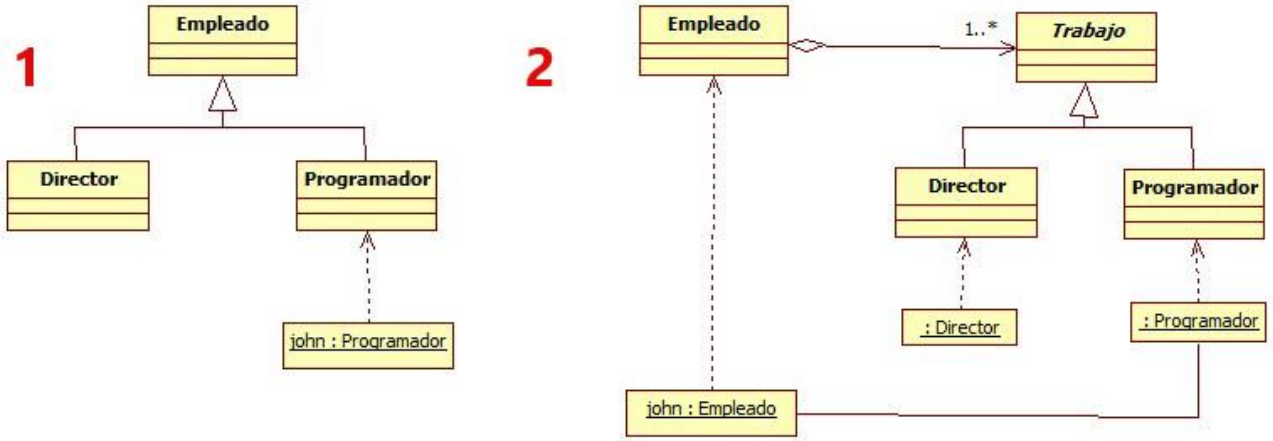
La herencia tiene ciertas características no deseables:

- Es la mayor forma de acoplamiento posible entre dos clases.
- El encapsulamiento es débil dentro de una jerarquía de clases. Los cambios en la clase base provocan cambios en las subclases.
- Es un tipo muy inflexible de relación. Las relaciones de herencia no se pueden cambiar en tiempo de ejecución. Cuando se necesita esto, se usa agregación.

Ejemplo (Herencia vs Agregación/Composición)

En el primer caso el objeto `:john` es `Programador`, pero si quisiera después pasar a ser `Director`, habría que crear un nuevo objeto `Director`, copiar todos los datos de `john:Programador` y luego eliminar el objeto `john:Programador` para mantener la consistencia de la aplicación.

En cambio, en el segundo caso, utilizando agregación se obtiene un modelo con semántica correcta, permitiendo cambiar a John a director en tiempo de ejecución. Para ello hay que cambiar el vínculo entre `john:Empleado - :Programador` a `john:Empleado - :Director`.



Herencia múltiple

Algunas veces es posible que se desee heredar de más de un padre. Esto es herencia múltiple y no es soportado en todos los lenguajes de programación orientados a objetos. Por ejemplo, Java y C# solamente permiten herencia simple.

Esta restricción de los lenguajes es importante porque una vez que aplicamos herencia a una clase, la subclase no va a poder heredar de otra si lo necesitamos. Suele sustituirse por el uso de interfaces.

La herencia no es la única opción

- **Delegación:** delegar comportamiento a otra clase cuando no quiere cambiar el comportamiento, pero no es su responsabilidad de sus objetos implementar ese comportamiento por sí mismos.
- **Composición:** puede reusar el comportamiento de una o más clases, y en particular de una familia de clases, con composición. El objeto posee completamente a los objetos compuestos y ellos no existen fuera del uso del objeto.
- **Agregación:** cuando se requieren los beneficios de la composición pero se utiliza el comportamiento de un objeto que existe más allá del objeto que lo uso.

Clase abstracta

Son clases que tiene algunos (o todos) sus métodos abstractos, o sea sin implementar. Esos métodos deben ser implementados por las clases hijas. Las clases abstractas no se pueden instanciar. Las clases que no son abstractas se denominan concretas.

Interfaz

Una interfaz especifica un conjunto de características públicas. La idea es separar la especificación de funcionalidad (la interfaz) de su implementación por un clasificador como una clase o subsistema. Una interfaz no se puede instanciar; simplemente declara un contrato que se puede realizar por cero o más clasificadores.

Herencia vs composición

Herencia: permite definir una implementación en términos de otra. A esto se lo denomina **reutilización de caja blanca**. Este término se refiere a la visibilidad con la herencia, ya que las interioridades de las clases padres suelen hacerse visibles a las subclases.

Composición: la funcionalidad se obtiene ensamblando o *componiendo* objetos para obtener funcionalidad más compleja. La composición se objetos requiere que los objetos a componer tengan interfaces bien definidas. A este tipo de reutilización se lo denomina **reutilización de caja negra**, porque los detalles internos de los objetos no son visibles.

HERENCIA	COMPOSICIÓN
Ventajas • Se define en tiempo de compilación. • Es sencilla de utilizar (se implementa directamente en el lenguaje de programación). • Fácil modificar la implementación que está siendo utilizada.	**Ventajas** • Se define en tiempo de ejecución (con objetos que hacen referencia a otros objetos). • Requiere objetos con interfaces bien diseñadas, dado que a los objetos se accede a través de sus interfaces. • No rompe el encapsulamiento (por lo del punto anterior, las interfaces). • Cualquier objeto puede ser reemplazado por otro en tiempo de ejecución siempre que sean del mismo tipo. • Las dependencias de implementación son menores. • Se mantiene el encapsulamiento y cada clase se centra en una tarea. • El comportamiento del sistema depende de las relaciones en vez de estar definido en las clases.
Desventajas • No se pueden cambiar las implementaciones heredades de las clases padre en tiempo de ejecución (se define en tiempo de compilación). • Rompe el encapsulamiento (lo detalles del padre son expuestos a las subclases). • Alto acoplamiento (los cambios en las clases padre obliga a cambiar las subclases).	

Herencia de clases vs herencia de interfaces

- **Clase**: la clase de un objeto define como se implementa el objeto.
- **Tipo**: el tipo de un objeto sólo se refiere a su interfaz (al conjunto de peticiones que puede responder).

Un objeto puede tener muchos tipos, y objetos de clases diferentes pueden tener el mismo tipo.

Herencia de clases (generalización): define la implementación de un objeto en términos de la implementación de otro objeto. Es un mecanismo para compartir código y representación.

Herencia de interfaces (realización): describe cuando se puede usar un objeto en lugar de otro.

Programar para interfaces, no para una implementación

La herencia de clases no es más que un mecanismo para extender la funcionalidad de una aplicación reutilizando la funcionalidad de las clases padres. Define rápidamente un nuevo tipo de objeto basándose en otro.

También es importante la capacidad de la herencia para definir familias de objetos con interfaces idénticas, al ser esto en lo que se basa el **polimorfismo**.

Cuando la herencia se usa con cuidado, todas las clases que derivan de una clase abstracta compartirán su interfaz. Esto implica que las subclases añaden o redefinen operaciones y no ocultan operaciones de la clase padre. Todas las subclases pueden entonces responder a las peticiones en la interfaz de su clase abstracta, convirtiéndose así todas ellas en subtipos de la clase abstracta.

Manipular los objetos solamente en términos de la interfaz definida por las clases abstractas tiene dos ventajas:

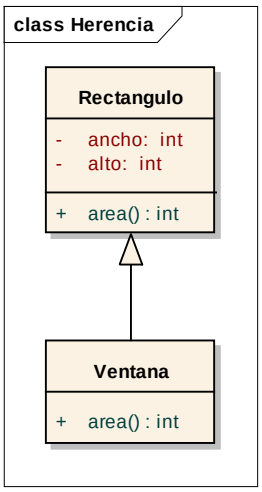
1. Los clientes no tienen que conocer los tipos específicos de los objetos que usan, basta que con éstos se adhieran a la interfaz que esperan los clientes.
2. Los clientes desconocen las clases que implementan dichos objetos; sólo conocen las clases abstractas que definen la interfaz.

Esto reduce las dependencias de implementación entre subsistemas, llevando al principio "*Programe para una interfaz, no para su implementación*". Es decir, no se deben declarar variables como instancias de clases concretas. En vez de eso, se ajustarán a la interfaz definida por una clase abstracta (o interfaz).

Los patrones de creación aseguran que el sistema se escriba en términos de interfaces, no de implementaciones.

Delegación

La delegación es un modo de lograr que la composición sea tan potente para la reutilización como lo es la herencia. Dos son los objetos encargados de tratar una petición: un objeto receptar delega operaciones a su delegado.



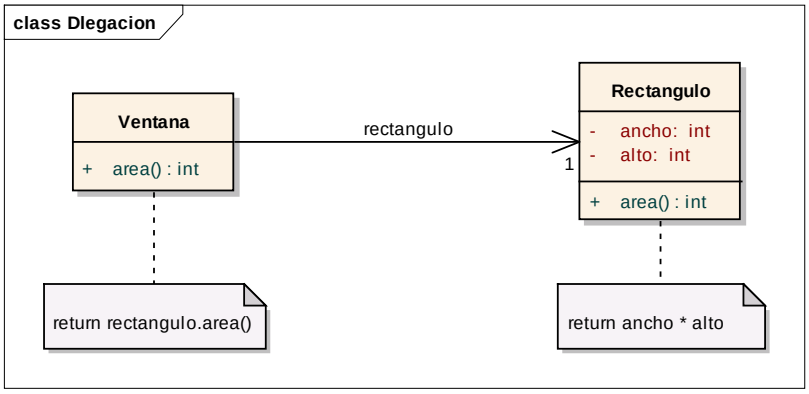
La principal ventaja de la delegación es que se hace que sea fácil combinar comportamientos en tiempo de ejecución y cambiar la manera en que éstos se combinan.

Se utiliza mejor cuando se quiere usar la funcionalidad de una clase tal cual es, sin cambiar su comportamiento para nada.

Es una alternativa a la herencia que no viola el principio de sustitución.

La delegación tiene un inconveniente: el software dinámico y altamente parametrizado es más difícil de entender que el estático. Hay también ineficiencias en tiempo de ejecución. La delegación es una buena elección de diseño solo cuando simplifica más de lo que complica.

Varios patrones de diseño utilizan la delegación, algunos son el State, Stretegy y Visitor.



Delegación vs. Colaboración

En la delegación, un objeto le delega la totalidad de la tarea a un objeto distinto para que la realice. Ejemplo: patrón State y Strategy.

En la colaboración, un objeto le pide a otro objeto que haga una parte de la tarea (este a su vez puede pedirle a otro y así sucesivamente, formando una "cadena"), entregándole un valor de retorno al objeto que invocó la tarea (y a través de la cadena si hay) hasta retornar un valor en el primer objeto. Ejemplo: cuando un gestor tiene que hacer algo, y pide a otro objeto que haga algo (y este a otros), pero siempre termina el valor de retorno en el gestor. Esto se ve claramente en el diagrama de comunicación.

PATRONES DE DISEÑO

Introducción

Diseñar software orientado a objetos es difícil, y aún lo es más diseñar software orientado a objetos reutilizable. Nuestro diseño debe ser específico del problema que estamos manejando, pero también lo suficientemente general para adecuarse a futuros requerimientos y problemas. También queremos evitar el rediseño, o al menos minimizarlo.

Lo que no hay que hacer es resolver cada problema partiendo de cero. Por el contrario, hay que reutilizar soluciones que ya han sido útiles en el pasado. Cuando se encuentra una solución buena, se usa una y otra vez.

Los patrones resuelven estos problemas concretos de diseño y hacen que los diseños orientados a objetos sean más flexibles, elegantes y reutilizables. Los patrones ayudan a los diseñadores a reutilizar buenos diseños al basar los nuevos diseños en la experiencia previa.

Cada patrón nomina, explica y evalúa un diseño importante y recurrente en los sistemas orientados a objetos. El objetivo, es representar esa experiencia de diseño de forma que pueda ser reutilizada de manera efectiva por otras personas.

- Los patrones de diseño hacen que sea más fácil reutilizar buenos diseños y arquitecturas.
- Los patrones de diseño nos ayudan a elegir alternativas de diseño que hacen que un sistema sea reutilizable, y a evitar fallas que dificultan dicha reutilización.
- Pueden mejorar la documentación y mantenimiento de sistemas existentes.
- Los patrones de diseño ayudan a lograr un buen diseño más rápidamente.

Qué es un patrón de diseño

Patrón: cada patrón describe un problema que ocurre una y otra vez en un determinado entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces.

Elementos de un patrón de diseño

1. **Nombre:** describe con una o dos palabras un problema de diseño junto con sus soluciones y consecuencias.
    - Incrementan el vocabulario de diseño: nos permite hablar con otros colegas, mencionarlos en documentación y tenerlos en cuenta. Más fácil pensar diseños y transmitirlos a otros.
    - Nos permiten diseñar con abstracción de un nivel más alto.
2. **Problema:** describe cuando aplicar el patrón. Explica el problema y su contexto.
3. **Solución:** describe los elementos que forman el diseño, sus relaciones, responsabilidades y colaboraciones.
    - No describe ningún diseño concreto.
    - Es como una plantilla que puede aplicarse en muchas situaciones diferentes.
4. **Consecuencias:** son los resultados así como las ventajas e inconvenientes de aplicar el patrón.
    - Son críticas para evaluar el patrón y entender los costos y beneficios de aplicarlo.
    - Suele relacionar el equilibrio entre el espacio y el tiempo.
    - Puede tratar asuntos de lenguajes e implementaciones.

Descripción de los patrones de diseño (plantilla de definición)

Cada patrón se divide en secciones de acuerdo a la siguiente plantilla:

- **Nombre del patrón y clasificación**
    - El nombre del patrón transmite su esencia.
- **Propósito**
    - Frase que responde a lo siguiente: ¿Qué hace este patrón? ¿En qué se basa? ¿Cuál es el problema concreto que resuelve?
- **También conocido como (sinónimos)**
    - Otros nombres
- **Motivación**
    - Escenario que ilustra un problema particular y cómo el patrón lo resuelve.
- **Aplicabilidad**
    - ¿En qué situaciones se puede aplicar? ¿Cómo se reconocen? Ejemplos de diseños que pueden mejorarse.
- **Estructura**
    - Diagrama de clases es ilustra la estructura.
- **Participantes**
    - Clases que participan y sus responsabilidades.
- **Colaboraciones**
    - Diagramas de interacción que muestran cómo colaboran los participantes.
- **Consecuencias**
    - ¿Cómo alcanza los objetivos el patrón? Ventajas e inconvenientes, alternativas.
- **Implementación**

    - Dificultades, trucos o técnicas que deberíamos tener en cuenta al aplicar el patrón.
- **Ejemplo de código**
    - El patrón implementado en algún lenguaje de programación como Java, C++, etc.
- **Usos conocidos**
    - Ejemplos del patrón en sistemas reales.
- **Patrones relacionados**
    - Patrones estrechamente relacionados con el patrón analizado. Diferencias, alternativas, etc.

¿A qué ayudan los patrones?

1. Encontrar los objetos apropiados.
2. Determinar la granularidad de los objetos.
3. Especificar interfaces de objetos.
4. Especificar implementaciones de objetos.
5. Favorecer reutilización.
6. Diseñar para el cambio.

Encontrar los objetos apropiados

Lo más complicado del diseño orientado a objetos es descomponer un sistema en objetos. Es difícil porque entran en juego muchos factores: encapsulamiento, granularidad, dependencia, flexibilidad, rendimiento, reutilización, etc.

Muchos objetos de un diseño proceden del modelo de análisis. Pero los diseños orientados a objetos suelen acabar teniendo clases que no tienen su equivalente en el mundo real. Algunas de ellos son de bajo nivel como los arrays, otras son de mucho más alto nivel, estas son clases de fabricación pura. Las abstracciones que surgen durante el diseño son fundamentales para lograr un diseño flexible.

Los patrones de diseño ayudan a identificar abstracciones menos obvias y los objetos que las expresan. Algunos son: Strategy y State.

Determinar la granularidad de los objetos

Los objetos pueden variar enormemente de tamaño y número. Pueden representar cualquier cosa. ¿Cómo decidimos entonces qué debería ser un objeto? Los patrones de diseño también se encargan de ésta cuestión. Ejemplos: Facade, Flyweight, Builder, etc.

Especificar las interfaces de los objetos

Cada operación declarada por un objeto especifica el nombre de la operación, los objetos que toma como parámetros y el valor de retorno de la operación. Esto se lo conoce como **signatura** o **firma**. Al conjunto de todas las firmas definidas por las operaciones de un objeto se le denomina **interfaz** del objeto. Dicha interfaz caracteriza el conjunto de peticiones que se puede enviar al objeto.

Las interfaces son fundamentales en el diseño orientado a objetos, ya que los objetos sólo se conocen a través de su interfaz. La interfaz de un objeto no dice nada acerca de su implementación (dos objetos con implementaciones diferentes pueden tener interfaces idénticas).

Objetos diferentes que soportan peticiones idénticas pueden tener distintas implementaciones de las operaciones que satisfacen esas peticiones. La asociación en tiempo de ejecución entre una petición a un objeto y una de sus operaciones es lo que se conoce como **enlace dinámico**.

El enlace dinámico significa que para enviar una petición no nos liga a una implementación particular hasta el tiempo de ejecución. Esto no permite escribir programas que esperen un objeto con una determinada interfaz, sabiendo que cualquier objeto que tenga la interfaz correcta aceptará la petición. El enlace dinámico nos permite sustituir objetos en tiempo de ejecución por otros que tengan la misma interfaz (**polimorfismo**). El polimorfismo simplifica

las defunciones de los clientes, desacopla unos objetos de otros y permite que varíen las relaciones entre ellos en tiempo de ejecución.

Los patrones de diseño nos ayudan a definir interfaces identificando sus elementos clave y los tipos de datos que se envían a la interfaz. Un patrón de diseño también puede decir qué no debemos poner en la interfaz. Ejemplo: Memento.

Los patrones de diseño también especifican relaciones entre interfaces. Ejemplo: Decorator, Proxy.

Especificar las implementaciones de los objetos

La implementación de un objeto queda definida por su clase. La clase especifica los datos, la representación interna del objeto y define las operaciones que puede realizar. *Definimos implementaciones a partir de otras (herencia).*

- "Programar para interfaces, no para una implementación.
- Herencia de clases vs Herencia de interfaces.
- No declarar variables de clases concretas sino abstractas (o interfaces).
- Patrones de creación permiten que un sistema esté basado en términos de interfaces y no en implementaciones.
- Favorece reutilización.
- Delegación.

Favorecer reutilización

- Herencia vs composición.
- Delegación.
- Tipos parametrizados (generics en Java). Ejemplo: `List<Integer> enteros; Set<Persona> personas;`

Diseñar para el cambio

La clave para maximizar la reutilización reside en anticipar nuevos requerimientos y cambios en los requerimientos existentes, y en diseñar los sistemas de manera que puedan evolucionar en consecuencia.

Un diseño que no tenga en cuenta el cambio sufre el riesgo de tener que ser rediseñado por completo en el futuro.

Los patrones de diseño ayudan a evitar esto al asegurar que un sistema pueda cambiar de formas concretas.

Toolkits

Un toolkit es un conjunto de clases relacionadas y reutilizables diseñadas para proporcionar funcionalidad útil de propósito general. Los toolkits no imponen un diseño particular en una aplicación, solo proporcionan funcionalidad que puede ayudar a que la aplicación haga su trabajo. Nos permiten evitar recodificar funcionalidad común. Los toolkits se centran en la reutilización de código.

Frameworks

Un framework es un conjunto de clases cooperantes que constituyen un diseño reutilizable para una clase específica de software.

El framework determina la arquitectura de nuestra aplicación. Definirá la estructura general, su partición en clases y objetos, las responsabilidades clave, como colaboran las clases y objetos y el hilo de control.

Los frameworks hacen hincapié en la reutilización del diseño frente a la reutilización de código. La reutilización a este nivel lleva a una **inversión de control** (IoC) entre la aplicación y el software en el que se basa.

Como resultado, no sólo se pueden construir aplicaciones más rápidamente, sino que las aplicaciones tienen estructuras parecidas, por lo que son más fáciles de mantener y resultan más consistentes para los usuarios. Por

otro lado, perdemos algo de libertad creativa, puesto que muchas decisiones de diseño ya han sido tomadas por nosotros.

Frameworks vs patrones

1. Los patrones de diseño son más abstractos que los frameworks. Los frameworks pueden plasmarse en código mientras que en los patrones solo los ejemplos pueden plasmarse en código.
2. Los patrones de diseño son elementos arquitectónicos más pequeños que los frameworks. Un framework contiene varios patrones de diseño, pero lo contrario nunca es cierto.
3. Los patrones de diseño están menos especializados que los frameworks. Los frameworks siempre tienen un dominio de aplicación concreto.

¿Cómo seleccionar un patrón de diseño?

- Considera de qué forma los patrones resuelven problemas de diseño.
- Lee la sección que describe el propósito de cada patrón.
- Estudia las interrelaciones entre patrones.
- Analiza patrones con el mismo propósito.
- Examina las causas de rediseñar.
- Considera que debería ser variable en tu diseño.

¿Cómo usar un patrón?

- Lee el patrón, todos sus apartados.
- Obtenga una visión global.
- Estudia la estructura, participantes y colaboraciones.
- Mira el ejemplo de código.
- Asocia a cada participante del patrón un elemento software de tu aplicación.
- Implementa las clases y métodos relacionados con el patrón.

Clasificación

		PROPÓSITO		
		Creación	*Estructural*	*Comportamiento*
ÁMBITO	*Clase*	**Factory Method**	Adapter (de clases)	Interpreter **Template Method**
	Objeto	**Abstract Factory** **Builder** Prototype **Singleton**	Adapter (de objetos) **Bridge** **Composite** Decorator **Facade** Flyweight **Proxy**	Chain of Responsability **Command** **Iterator** Mediator **Memento** **Observer** **State** **Strategy** Visitor

Patrones de creación

Los patrones de diseño de creación abstraen el proceso de creación de instancias (de objetos). Ayudan a hacer un sistema independiente de cómo se crean, se componen y se representan sus objetos.

Dos tipos:

- **Patrón de creación de clase:** usa la herencia para cambiar la clase que es instanciada.
- **Patrón de creación de objeto:** delega la creación de la instancia a otro objeto.

Hay dos temas recurrentes en estos patrones:

- Todos encapsulan el conocimiento sobre las clases concretas que usa el sistema.
- Todos ocultan como se crean y se asocian las instancias de estas clases.

Todo lo que el sistema conoce de los objetos son sus interfaces. Por lo tanto, da mucha flexibilidad a qué es lo que se crea, quién lo crea y cuando.

Patrones de creación

1. **Abstract Factory:** proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.
2. **Factory Method:** define una interfaz para crear un objeto, pero deja que las subclases decidan qué clase instanciar. Delega en las subclases la creación de objetos.
3. **Singleton:** garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a ella.
4. **Builder:** separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.
5. **Prototype:** especifica los tipos de objeto a crear por medio de una instancia prototípica, y crea nuevos objetos copiando de este prototipo.

Patrones de estructura

Los patrones estructurales se ocupan de cómo se combinan las clases y objetos para formar estructuras más grandes y complejas.

Dos tipos:

- **Patrones estructurales de clase:** hacen uso de la herencia ara componer interfaces o implementaciones.
- **Patrones estructurales de objeto:** describen formas de componer objetos para obtener nueva funcionalidad. Hacen uso de la composición.

Patrones de estructura

1. **Adapter:** convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permite que cooperen clases que de otra forma no podrían por tener interfaces incompatibles.
2. **Bridge:** desacopla una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.
3. **Composite:** combina (compone) objetos en estructuras de árbol para representar jerarquías de todo-parte. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
4. **Decorator:** añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender funcionalidad.
5. **Facade:** proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.
6. **Flyweight:** usa comportamiento para permitir un gran número de objetos de granularidad fina de forma eficiente.
7. **Proxy:** proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

Patrones de comportamiento

Los patrones de comportamiento tienen que ver con algoritmos y con la asignación de responsabilidades a objetos. No sólo describen patrones de clases y objetos, sino también patrones de comunicación entre ellos.

Dos tipos:

- **Patrones de comportamiento de clase:** usan la herencia para distribuir el comportamiento entre clases.

- **Patrones de comportamiento de objeto:** usan la composición de objetos en vez de la herencia.

Las claves son cooperaciones entre objetos para realizar tareas complejas que por sí solos no podrían realizar, reduciendo la dependencia entre objetos (Iterator, Observer, etc.); y, asociar comportamiento a objetos e invocar su ejecución (Command, Strategy, etc.).

Patrones de comportamiento

1. **Chain of responsability:** evita acoplar el emisor de una petición a su receptor, dando a más de un objeto la posibilidad de responder la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que ésta sea tratada por algún objeto.
2. **Command:** encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones.
3. **Interpreter:** dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.
4. **Iterator:** proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer a su representación interna.
5. **Mediator:** define un objeto que encapsula cómo interactúan una serie objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.
6. **Memento:** representa y externaliza el estado interno de un objeto sin violar el encapsulamiento, de forma que éste pueda volver a dicho estado más tarde.
7. **Observer:** define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él.
8. **State:** permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.
9. **Strategy:** define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.
10. **Template Method:** define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.
11. **Visitor:** representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

(Para más detalles ver el libro de patrones de diseño de Gamma y/o las filminas de la Meles)

DISEÑO DE PERSISTENCIA (BASES DE DATOS)

Introducción

La necesidad de usar una base de datos proviene de la capacidad limitada de la memoria primaria (RAM). Las bases de datos se almacenan frecuentemente sobre **almacenamiento secundario** (discos rígidos), proveyendo maneras eficientes de acceder a los datos. Otra necesidad proviene de almacenar objetos mayores que una ejecución de programa (que el objeto sobreviva a la ejecución que lo creó). Esta capacidad se llama **persistencia**. La persistencia frecuentemente significa que los objetos se copian desde una memoria primaria rápida y volátil a una memoria secundaria, lenta y persistente.

DBMS

Necesitamos un sistema de gestión de bases de datos (DBMS) que maneje el almacenamiento en la base de datos. El programador solo quiere una vista lógica de la base de datos y no tomar decisiones sobre cómo se hace el almacenamiento físico.

Un DBMS debería proveer:

- **Concurrencia:** permite a múltiples usuarios trabajar con una base de datos común simultáneamente.
- **Recuperación:** si ocurre una falla (hardware o software), el DBMS debería ser capaz de volver la base de datos a un estado uniforme de datos.
- **Facilidad de consulta:** el DBMS debería soportar una manera fácil de acceso a los datos en la base de datos.

Propiedades típicas que indican necesidad de un DBMS

- La información necesita ser persistente.
- Más de una aplicación compartiendo (parte de) los datos.
- La estructura de la información con un gran número grande de instancias.
- Búsquedas complejas en la estructura de información.
- Generación avanzada de informes desde la información almacenada.
- Manipulación de transacciones de usuario.
- Un registro para el reinicio de sistema.

Qué objetos necesitan persistencia

Los modelos de análisis y de requerimientos pueden estudiarse para decidir qué objetos van a ser persistentes. Los objetos de entidad son los candidatos importantes para el almacenamiento persistente.

Modelos de datos

Los DBMSs han evolucionado mediante un número de generaciones incluyendo modelos de datos jerárquicos, de red, relacionales y orientados a objetos. El modelo de datos dominante en aplicaciones industriales en la actualidad es el modelo relacional.

Modelo relacional y problema de impedancia

En una base de datos relacional, los datos se almacenan en tablas. Los tipos a ser usados en las tablas son mayormente los tipos primitivos (caracteres, enteros, reales, etc.). Esto trae problemas a la hora de almacenar objetos.

Problema de impedancia

Se refiere al problema de guardar objetos en un modelo relacional:

- En una base de datos relacional solo se guardan los datos, no el comportamiento.
- En una base de datos relacional solo se pueden almacenar datos primitivos y no objetos complejos.

Como se trabaja en un lenguaje de programación orientado a objetos, toda nuestra información se almacena en objetos, por lo tanto, necesitamos transformar nuestra estructura de información de objeto a una estructura orientada a tablas (modelo relacional). Este problema se denomina **problema de impedancia**.

El problema principal viene con los objetos complejos (objetos que tienen comportamiento, y poseen atributos que además de poder ser primitivos, pueden ser complejos, o sea atributos que hacen referencia a otros objetos). Todos estos tipos deben convertirse a los tipos de datos primitivos del RDBMS (Modelo de Bases de Datos Relacional).

El problema de impedancia trata aún otro problema: crea un fuerte acoplamiento entre la aplicación y el DBMS. Para ello, pocas partes de nuestro sistema (lo mínimo posible) deberían saber sobre la interface del DBMS.

Un tercer problema es cómo expresar la herencia en la base de datos.

Otros problemas que provienen incluyen como almacenar operaciones, la vista de transacción, bloqueo de datos durante transacciones más largas y distribución.

En resumen:

- Problema al almacenar objetos complejos (comportamiento y atributos con referencia a otros objetos). Los datos deben ser primitivos.
- Problema para representar la herencia.
- Fuerte acoplamiento entre la aplicación y el DBMS.

Objetos en tablas

La primera cosa para hacer es determinar que clases y variables de la clase deben almacenarse en la base de datos. Cada una de estas clases entonces será representada por una tabla (por lo menos una) en la base de datos.

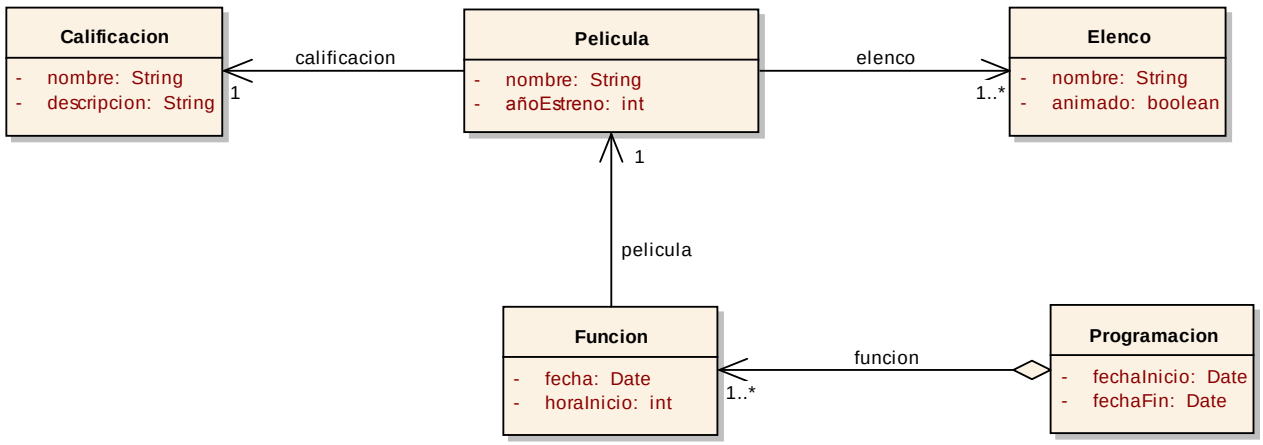
Una clase se mapea en tablas de la siguiente manera:

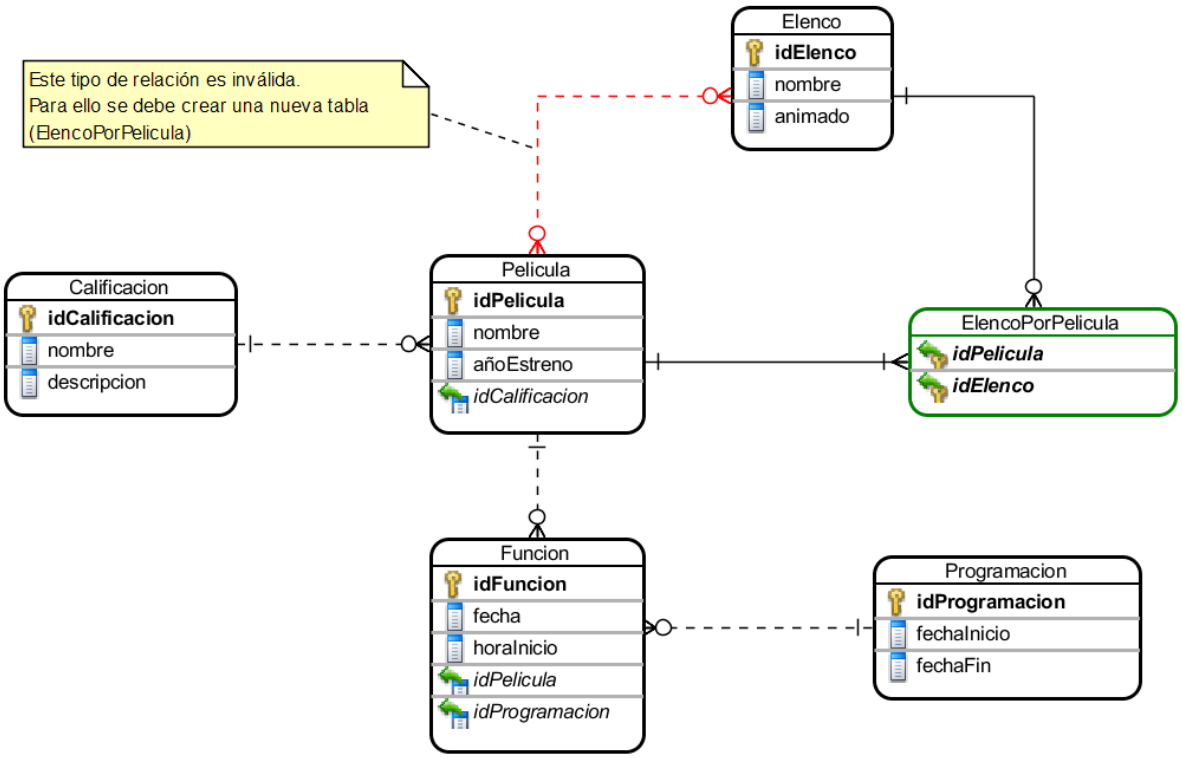
1. Asignar una tabla para la clase.
2. Cada atributo primitivo se transformará en una columna en la tabla. Si el atributo es complejo, agregamos una tabla adicional para el atributo, o distribuimos el atributo en varias columnas de la tabla de la clase.
3. La columna de la clave primaria será el identificador único de la instancia. El identificador debe ser preferentemente invisible al usuario y generadas automáticamente por máquina.
4. Cada instancia de la clase será representada por una fila en la tabla.
5. Cada asociación de conocimiento con una cardinalidad mayor que 1 (0..N por ejemplo)se transformará en una nueva tabla. Esta nueva tabla conectará las tablas que representan los objetos que van a ser asociados mediante el identificador de la clave primaria. En algunos casos, podemos tener la relación representada como una columna (atributo) en la tabla del objeto asociado.

Normalización

La normalización de la base de datos consiste en el hecho de eliminar redundancia en la base de datos y evitar ciertas anomalías en la actualización. En la mayoría de los casos es suficiente con alcanzar la tercera forma normal (3FN) para el diseño de bases de datos. Un diseño de bases de datos basado en un modelo de objetos acabará normalmente en 3FN desde el comienzo. La 3FN afirma que, si y solo si para todas las veces, cada fila consiste de un único objeto identificador junto con un número de valores de atributos mutuamente independientes, entonces la tabla está en 3FN.

Cuando la base de datos está normalizada, el problema de la baja performance ocurre. La baja performance puede resolverse en muchos casos con tablas de indexado especial. Si es no es posible, el problema verdadero es “desnormalizar” la base de datos para aumentar su performance. En este caso podemos tener problemas de redundancia en la base de datos.



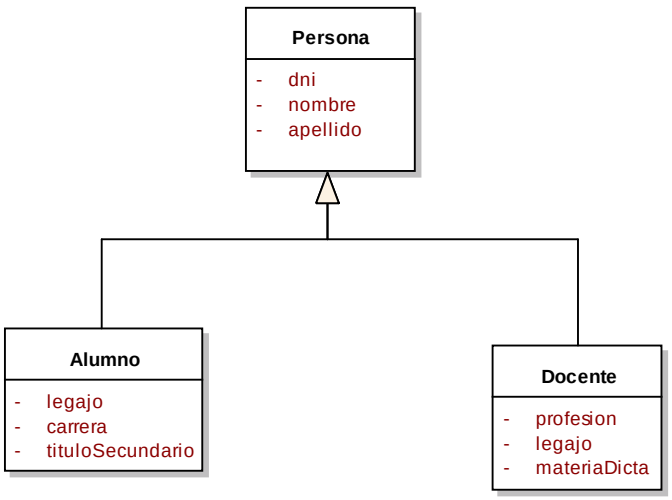


Herencia

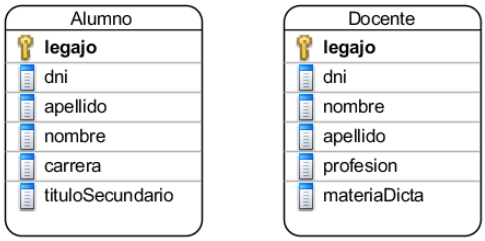
Si las clases se heredan unas a otras, hay principalmente tres maneras diferentes de resolver esto:

1. Tablas únicas para los objetos hijos. Se copian todos los atributos de la clase padre. Ninguna tabla representa la clase abstracta.
    - Es más rápida ya que los datos están en una sola tabla (no hay que hacer JOIN)
    - El tamaño de la base de datos aumenta ya que se duplican columnas.
    - Puede traer problemas de consistencia y redundancia.
2. La clase abstracta está en su propia tabla y las tablas de los hijos hacen referencia a ella.
    - Reduce la redundancia.
    - Requiere hacer JOINs que pueden reducir la performance.
3. Eliminar hijos y poner los atributos de todas las clases hijas en una sola tabla (no recomendado).

Ejemplo

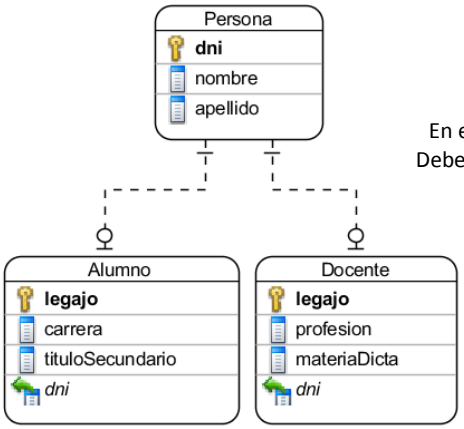


Según el primer caso quedaría:



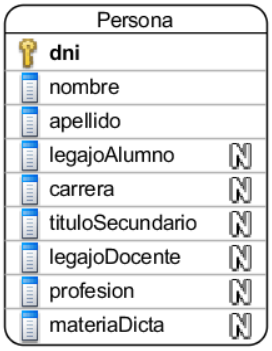
Aquí siempre hay que mantener la consistencia en caso de que un Alumno sea también Docente.

Según el segundo caso:



En este caso siempre hay que hacer JOIN. Deberíamos usar un DBMS que tenga buena performance en JOIN.

Según el tercer caso (no debería ser usado):



Si se quiere almacenar un alumno se dejan los datos de docente en null o viceversa.

Materialización y Desmaterialización

- **Materialización:** una o varias filas de tablas (registros) de la base de datos son convertidos a objetos en memoria.
  - Materialización inmediata: recupera los objetos al instante.
  - Materialización perezosa: recupera los objetos cuando se los necesita realmente.
- **Desmaterialización:** proceso inverso, un objeto se graba en la base de datos en forma de uno o varios registros (filas) de tablas.

DBMSs Orientados a Objetos

La idea de un DBMS orientado a objetos (ODBMS) es almacenar a los objetos como tales. De esta manera no tenemos que realizar ninguna unión lenta para conseguir el acceso a un objeto específico.

La mayoría de los ODBMSs comerciales no usan un lenguaje de manipulación de datos específico para almacenar y recobrar información, sino que usan el lenguaje de programación directamente. O sea que no es necesaria una interfaz especifica con el DBMS. Así el problema de impedancia se elimina completamente. Esto también significa que el diseño de la base de datos se integra durante el análisis y el diseño de la aplicación.

Qué criterios debe cubrir un DBMS orientado a objetos

1. **Objetos complejos:** debería soportar la noción de objetos complejos.
2. **Identidad de objetos:** cada objeto debe tener una identidad independientemente de sus valores internos.
3. **Encapsulamiento:** debe soportar el encapsulamiento de datos y comportamiento de objetos.
4. **Tipos o clases:** debería soportar un mecanismo de estructuración, en forma de tipos o clases.
5. **Jerarquía:** debería soportar la noción de herencia.
6. **Ligadura tardía:** debería soportar sobreescritura y ligadura tardía.
7. **Completitud:** el lenguaje de manipulación debería ser capaz de expresar cada función calculable.
8. **Extensibilidad:** debería ser posible agregar nuevos tipos.

Dado que los DBMSs relacionales se han usado desde hace mucho tiempo, estos productos son más maduros y también tienen un soporte extensivo a muchos puntos complejos incluido en el uso de bases de datos.

Ventajas de un DBMS orientado a objetos

- Los objetos se almacenan en la base de datos como tales.
- No se necesita ninguna conversión del tipo de sistema de DBMS; las clases definidas por el usuario se usan como tipos en el DBMS.
- El lenguaje del DBMS puede integrarse con un lenguaje de programación orientado a objetos.

Diseño de persistencia

Un **esquema de persistencia** es un conjunto reutilizable (y, generalmente expansible) de clases que prestan servicios a los objetos persistentes. Por lo general, un esquema de persistencia debe traducir los objetos a registros y guardarlos en una base de datos; después traduce los registros a objetos cuando los recupera de una base.

Frameworks de persistencia

- Es un conjunto extensible de clases e interfaces que colaboran para proporcionar servicios de la parte central e invariable de un subsistema lógico.
- Contiene clases concretas y (especialmente) abstractas que definen las interfaces a las que ajustarse, interacciones de objetos en las que participar, y otras variantes.
- Pueden requerir que el usuario del framework defina subclases del framework para utilizar, adaptar y extender los servicios del framework.
- Tiene clases abstractas que podrían contener tanto métodos abstractos como concretos.
- Confía en el **Principio de Hollywood** "*No nos llame, nosotros lo llamaremos*".
- Ofrecen un alto grado de reutilización, mucho más que con clases individuales.

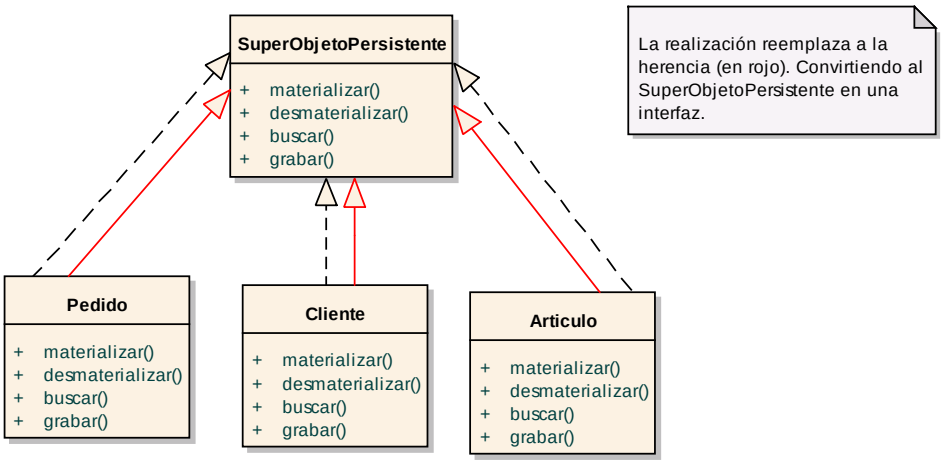
Debería proporcionar funciones para:

- Almacenar y recuperar objetos en un mecanismo de almacenamiento persistente.
- Confirmar y deschacer (*commit* y *rollback*) las transacciones.

Súper Objeto Persistente

Dado que cada objeto que debe almacenarse en la base de datos debe tener una funcionalidad de persistencia común, creamos un bloque abstracto que tiene esta funcionalidad. Las clases en este bloque abstracto forman un framework. Este bloque es heredado por todos los bloques que deben poder almacenar información en la base de datos y el framework se especializa para cada bloque particular.

Se corresponde con enfoque de **correspondencia directa**.

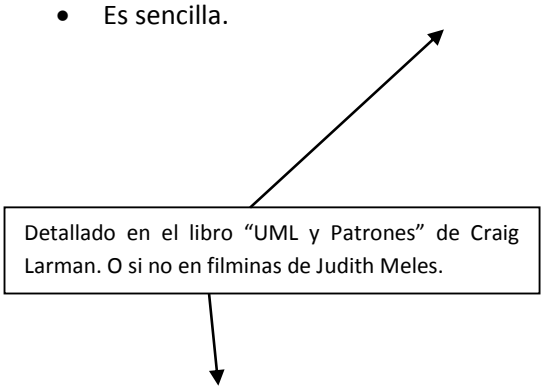


Ventajas

- Es rápida.
- Es sencilla.

Desventajas

- Como la mayoría de los lenguajes de programación OO sólo permiten herencia simple, impide que las clases hereden de otra. Esto se soluciona mediante una realización (implementación de interfaz) haciendo al SuperObjetoPersistente una interfaz.
- Se abren las clases de domino para darles una responsabilidad que naturalmente no les corresponde (la persistencia, hacen SELECT, INSERT, UPDATE, etc.). Esto disminuye la cohesión.
- Genera fuerte acoplamiento.



Esquema de persistencia

Ideas a desarrollar

1. **Correspondencia (*mapping*):** entre clases y tablas, ente atributos y campos. Es decir, correspondencia de esquemas.
2. **Identidad de objeto:** identificador único que vincula objetos con registros evitando duplicados.
3. **Materialización y desmaterialización:** transformar registros en objetos y objetos en registros.
4. **Conversor de bases de datos:** es el responsable de la materialización y desmaterialización de los objetos.
5. **Caché:** almacén de objetos materializados en esta memoria por cuestiones de rendimiento.
6. **Estado de transacción de los objetos:** el estado de los objetos para saber si se modificaron en función de su estado almacenado.
7. **Operaciones de transacción:** confirmar y deshacer (*commit* y *rollback*).
8. **Materialización perezosa:** no todos los objetos se materializan a la vez, se hace bajo demanda, es decir, cuando se los necesita.
9. **Proxies virtuales:** referencia inteligente utilizada para la materialización perezosa.

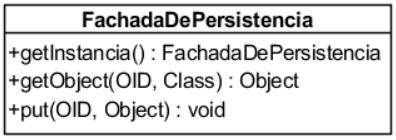
Patrón Identificador de Objetos

- Es necesario contar con una forma consistente de relacionar objetos con registros y asegurar que la materialización repetida no de cómo resultado objetos duplicados.
- El patrón propone asignar un identificador de objeto (OID, *Object ID*) a cada registro y objeto (o proxy de objeto).
- *Sirve para identificar al objeto (al registro que los representa) en una tabla sin equivocación.*

Acceso al servicio de persistencia mediante Fachada

- **Patrón fachada:** proporciona una interfaz uniforme a un subsistema.
- Se necesita una operación para recuperar un objeto dado un OID.

- También se necesita conocer el tipo del objeto que se va a materializar, por tanto, se debe proporcionar el tipo de la clase a la que pertenece el objeto.
- La fachada no hace el trabajo, solo lo delega en objetos del subsistema.



Patrón Conversor (*Mapper*) o Intermediario (*Broker*)

- Enfoque de **correspondencia indirecta.**
- Utiliza objetos para establecer la correspondencia con los objetos persistentes.
- Propone crear una clase responsable de materializar y desmaterializar los objetos.
- Se agrega un intermediario por cada clase de dominio (genera muchas clases, el doble).

Materialización con el método plantilla

- La estructura básica para materializar un objeto es:

```
if (objeto está en caché)
      return objeto
else
      crear objeto a partir de su representación en el almacenamiento
      guardar objeto en caché
      return objeto
```

- El punto de variación es la manera de crear el objeto a partir del almacenamiento.
- *Con el patrón Template Method encapsulamos en una sola clase el comportamiento común de la materialización.*

Patrón Gestión de Caché

- Es conveniente mantener los objetos materializados en una caché local para mejorar el rendimiento de y dar soporta a las operaciones de gestión de transacciones.
- Al materializar los objetos, quedan en la caché con su OID como clave.
- El conversor primero busca en la caché para evitar materializaciones innecesarias.

Las seis cachés son:

1. **Caché Limpia y Nueva**. Objetos nuevos, sin modificaciones.
2. **Caché Limpia y Vieja**. Objetos viejos que se materializan a partir de una base de datos, sin modificaciones.
3. **Caché sucia y nueva**. Objetos nuevos, modificados.
4. **Caché sucia y vieja**. Objetos viejos que se materializaron de una base de datos y que fueron modificados.
5. **Caché Eliminar Nueva**. Objetos nuevos a eliminar.
6. **Caché Eliminar Vieja**. Objetos viejos que se materializaron a partir de una base de datos y que deben ser eliminados.

Estados transaccionales y el Patrón State

- Los objetos persistentes pueden insertarse, eliminarse o modificarse.
- Operar sobre un objeto persistente no provoca una actualización inmediata sobre la base de datos, más bien debe ejecutar una operación commit explícita.
- Además considerar que la respuesta a una operación depende del estado de la transacción del objeto.

He aquí los posibles estados relevantes de la transacción:

1. **Limpio y nuevo**. Objetos nuevos, sin modificaciones.
2. **Limpio y viejo**. Objetos viejos materializados a partir de una base de datos, sin modificaciones.
3. **Sucio y nuevo**. Objetos nuevos, sin modificaciones.
4. **Sucio y viejo**. Objetos viejos materializados a partir de una base de datos, con modificaciones.
5. **Eliminar nuevo**. Objetos nuevos que deben ser eliminados.
6. **Eliminar viejo**. Objetos viejos que fueron materializados a partir de una base de datos y que deben ser eliminados.

Ejemplo

Un objeto "sucio viejo" debería actualizarse en la base de datos. Un "viejo limpio" no debería hacer nada porque no ha cambiado.

Diseño de una transacción con Patrón Command

- **Transacción:** unidad de trabajo (conjunto de tareas) cuya terminación es atómica (se realizan todas las tareas o ninguna).
- En los servicios de persistencia se incluyen: inserción, actualización, eliminación de objetos.
- Una transacción puede incluir por ejemplo: dos inserciones, una actualización, y tres eliminaciones.
- Las tareas necesitan que se ordenen justo antes de la ejecución.
- La idea clave es representar cada tarea o acción de la transacción como un objeto con un método *ejecutar()* polimórfico.

Lo que se busca con el patrón Command es crear un objeto por cada operación a realizar en la transacción, así se puede crear una cola de operaciones (cola de objetos) permitiendo si es necesario hacer un rollback (deshacer los cambios).

Materialización perezosa mediante un Proxy Virtual

- A veces es conveniente diferir la materialización de los objetos hasta que sea absolutamente necesario, por cuestiones de rendimiento.
- La materialización diferida de objetos se denomina **materialización perezosa**.
- Esta materialización se puede implementar utilizando un Proxy Virtual.
- Un Proxy Virtual es un proxy para otro objeto (el sujeto al que se quiere acceder realmente) que materializa el sujeto real cuando se referencia por primera vez.
- Este diseño se basa en la suposición de que los proxies conocen el OID de sus sujetos reales, y cuando se requiere la materialización, se utiliza el OID para ayudar a identificar y recuperar el sujeto real.

WORKFLOW DE IMPLEMENTACIÓN

Introducción

En la implementación se comienza con el resultado del diseño e implementamos el sistema en términos de componentes, es decir, archivos de código fuente, scripts, archivos de código binario, ejecutables y similares.

El propósito fundamental de la implementación es desarrollar la arquitectura el sistema como un todo.

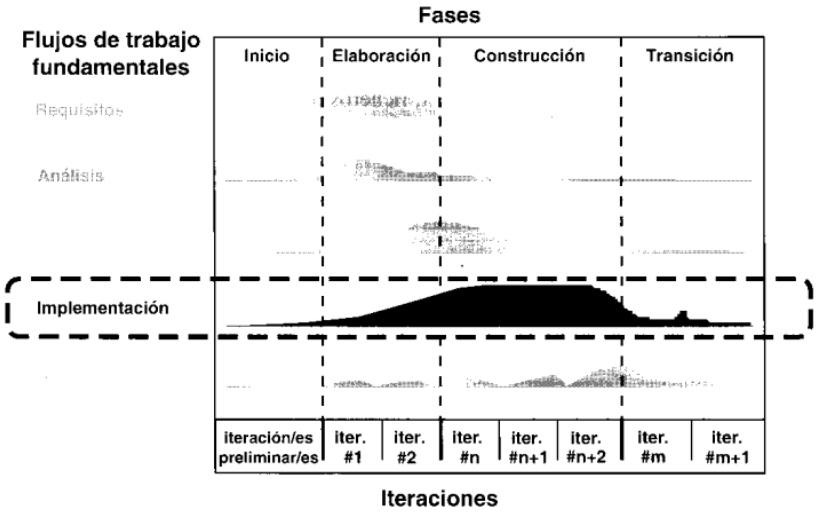
El resultado principal de la implementación es el modelo de implementación.

Propósitos de la implementación

- Planificar las integraciones de sistema necesarias en cada iteración. Se sigue un enfoque incremental, dando lugar a un sistema que se implementa en una sucesión de pasos pequeños y manejables.
- Distribuir el sistema asignando componentes ejecutables a nodos en el diagrama de despliegue.
- Implementar las clases y subsistemas encontrados durante el diseño.
- Probar los componentes individualmente, y a continuación integrarlos compilándolos y enlazándolos a uno o más ejecutables.

El papel de la implementación en el ciclo de vida del software

La implementación es el centro durante las iteraciones de construcción, aunque también se lleva a cabo trabajo de implementación durante la fase de elaboración, para crear la línea base ejecutable de la arquitectura, y durante la fase de transición, para tratar defectos tardíos.



Artefactos de la implementación

1. Modelo de implementación.
2. Componente.
3. Subsistema de implementación.
4. Interfaz.
5. Descripción de la arquitectura (vista del modelo de implementación).
6. Plan de integración de construcciones.

Modelo de implementación

El modelo de implementación describe cómo los elementos del modelo de diseño, como las clases, se implementan en términos de componentes, como ficheros de código fuente, ejecutables, etc. Describe también cómo se organizan los componentes de acuerdo con los mecanismos de estructuración y modularización disponibles en el entorno de implementación y en el lenguaje o lenguajes de programación utilizados, y cómo dependen los componentes unos de otros.

Componente

Un componente es el empaquetamiento físico de los elementos de un modelo, como son las clases en el modelo de diseño. Algunos estereotipos estándar son:

- <<executable>> es un programa que puede ser ejecutado en un nodo.
- <<file>> es un fichero que contiene código fuente o datos.
- <<library>> es una librería estática o dinámica.

- <<table>> es una tabla de base de datos.
- <<document>> es un documento.

Subsistema de implementación

Los subsistemas de implementación proporcionan una forma de organizar los artefactos del modelo de implementación en trozos más manejables. Puede estar formado por componentes, interfaces, y otros subsistemas (recursivamente).

Un subsistema de implementación se manifiesta a través de un "mecanismo de empaquetamiento" concreto en un entorno de implementación determinado, como

- Un paquete en Java.
- Un proyecto es Visual Basic.
- Un directorio de ficheros en un proyecto de C++.
- Un paquete en una herramienta de modelado como Rational Rose.

La semántica de la noción de *subsistema de implementación* se refinará ligeramente cuando se manifieste en un entorno de implementación determinado.

Los subsistemas de implementación están muy relacionados con los subsistemas de diseño en el modelo de diseño. Los subsistemas de implementación deberían seguir la traza uno a uno de sus subsistemas de diseño correspondientes.

- El subsistema de implementación debería definir dependencias análogas hacia otros subsistemas de implementación o interfaces.
- El subsistema de implementación debería proporcionar las misma interfaces.
- El subsistema de implementación debería definir qué componentes o, recursivamente, qué otros subsistemas de implementación dentro del subsistema deberían proporcionar las interfaces proporcionales por el subsistema.

Interfaz

Utilizadas en el modelo de implementación para especificar las operaciones implementadas por componentes y subsistemas de implementación.

Un componente que implementa (y por tanto proporciona) una interfaz ha de implementar correctamente todas las operaciones definidas por la interfaz. Un subsistema de implementación que proporciona una interfaz tiene que también que contener componentes que proporcionen la interfaz u otros subsistemas (recursivamente) que proporcionan la interfaz.

Descripción de la arquitectura (vista del modelo de implementación)

La descripción de la arquitectura contiene una **visión del modelo de implementación,** el cual representa sus artefactos significativos arquitectónicamente.

Artefactos considerados significativos para la arquitectura en el modelo de implementación:

- La descomposición del modelo en subsistemas, sus interfaces y las dependencias entre ellos.
- Componentes clave, como los componentes que siguen la traza de las clases de diseño significativas arquitectónicamente, los componentes ejecutables y los componentes que son generales, centrales, que implementan mecanismos de diseño genéricos de los que dependen muchos otros componentes.

Plan de integración de construcciones

Es importante construir el software incrementalmente en pasos manejables, de forma que cada paso dé lugar a pequeños problemas de integración o prueba. El resultado de cada paso es llamado "construcción", que es una

versión ejecutable del sistema. Cada construcción es entonces sometida a pruebas de integración antes de que se cree ninguna otra construcción. Para prepararse ante un fallo de una construcción se lleva un control de versiones de forma que se pueda volver atrás a una construcción anterior.

Cada iteración resultara en al menos una construcción. Sin embargo, la funcionalidad a ser implementada en una iteración determinada es a menudo demasiado compleja para ser integrada en una sola construcción. En lugar de esto, puede que se cree una secuencia de construcciones dentro de una iteración, cada una de las cuales representará un paso manejable en el que se hace un pequeño incremento al sistema.

Un **plan de integración de construcciones** describe la secuencia de construcciones necesarias en una iteración. Describe lo siguiente para cada construcción:

- La funcionalidad que se espera que sea implementada en dicha construcción.
- Las partes del modelo de implementación que están afectadas por la construcción.

Trabajadores de la implementación

1. Arquitecto.
2. Ingeniero de componentes.
3. Integrador de sistemas.

Arquitecto

El arquitecto es responsable de la integridad del modelo de implementación y asegura que el modelo como un todo es correcto, completo y legible.

El modelo es correcto cuando implementa la funcionalidad descrita en el modelo de diseño y en los requerimientos adicionales, y solo esa funcionalidad.

El arquitecto es responsable también de la arquitectura del modelo de implementación, es decir, de la existencia de sus partes significativas arquitectónicamente.

Un resultado importante de la implementación es la asignación de componentes ejecutables a nodos. El arquitecto es responsable de esta asignación, la cual se representa en la vista de la arquitectura del modelo de despliegue.

Ingeniero de componentes

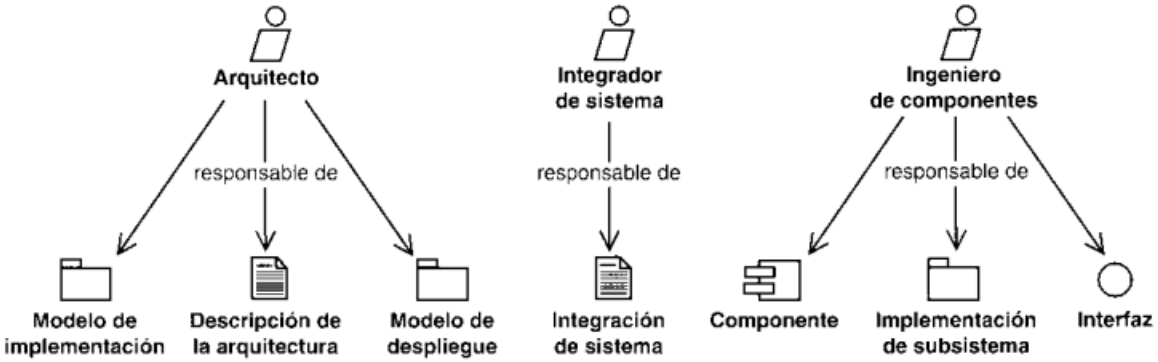
El ingeniero de componentes define y mantiene el código fuente de uno o varios componentes, garantizando que cada componente implementa la funcionalidad correcta.

A menudo, el ingeniero de componentes también mantiene la integridad de uno o varios subsistemas de implementación. Ya que los subsistemas de implementación siguen la traza uno a uno a los subsistemas de diseño., necesita garantizar que los contenidos de los subsistemas de implementación son correctos, que sus dependencias con otros subsistemas o interfaces son correctas y que implementan correctamente las interfaces que proporciona.

Un ingeniero de componentes debería, diseñar e implementar las clases bajo su responsabilidad.

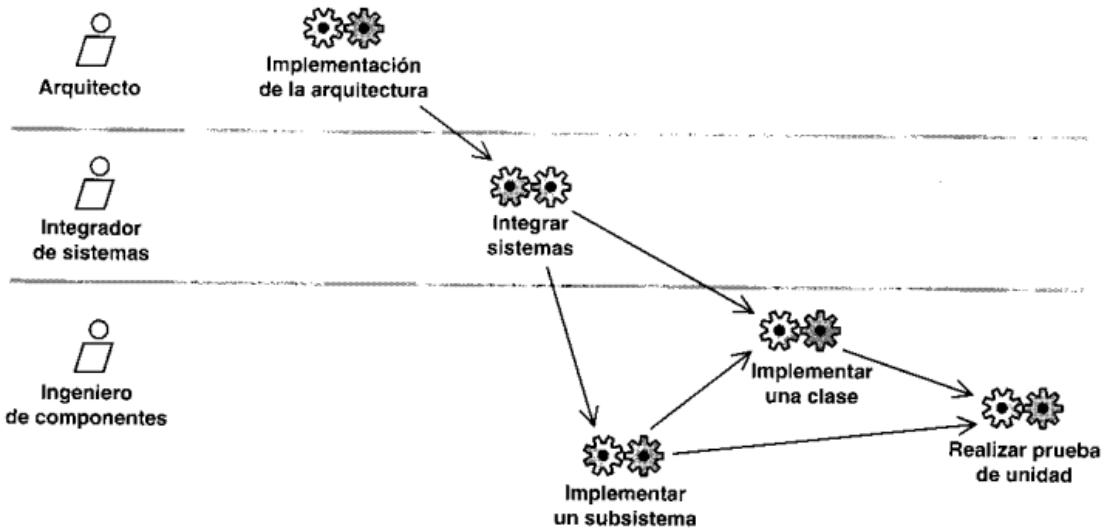
Integrador de sistemas

Entre sus responsabilidades se incluye planificar la secuencia de construcciones necesarias en cada iteración y la integración de cada construcción cuando sus partes han sido implementadas. La planificación da lugar a un plan de integración de construcciones.



Flujo de trabajo (actividades)

1. Este proceso es iniciado por el arquitecto esbozando los componentes clave en el modelo de implementación.
2. El integrador de sistemas planea las integraciones de sistema necesarias en la presente iteración como una secuencia de construcciones. Para cada construcción, el integrador de sistemas describe la funcionalidad que debería ser implementada y que partes del modelo de implementación se verán afectadas.
3. Los ingenieros de componentes implementan los requerimientos sobre los subsistemas y componentes en la construcción.
4. Los componentes son probados y pasados al integrador de sistemas para su integración. El integrador de sistemas integra los nuevos componentes en una construcción y pasa a los ingenieros de pruebas de integración para llevar a cabo pruebas de integración.
5. Los desarrolladores inician la implementación de la siguiente construcción, tomando en consideración los defectos de la construcción anterior.



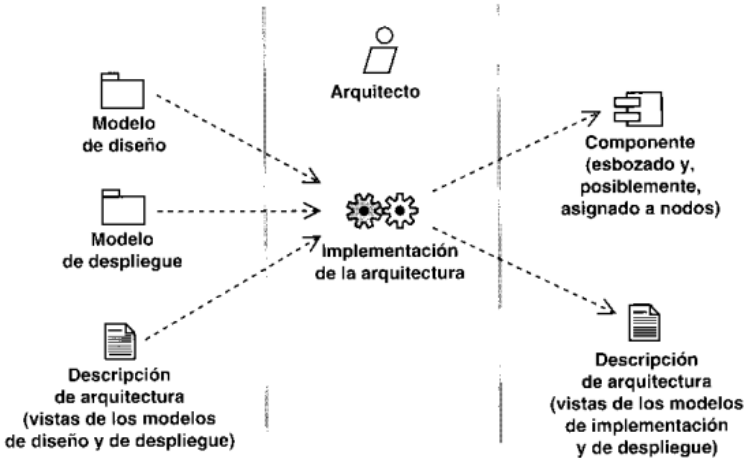
Implementación de la arquitectura (por arquitecto)

Es esbozar el modelo de implementación y su arquitectura mediante:

- La identificación de componentes significativos arquitectónicamente, tales como componentes ejecutables.
- La asignación de componentes a los nodos en las configuraciones de redes relevantes.

La identificación de los subsistemas de implementación y sus interfaces es más o menos trivial y por lo tanto no se trata aquí (traza uno a uno con los subsistemas de diseño y proporcionan las mismas interfaces).

Durante esta actividad, arquitecto mantiene, refina y actualiza la descripción de la arquitectura y las vista de la arquitectura de los modelos de implementación y de despliegue.



Integrar el sistema (por integrador de sistemas)

Los objetivos de la integración del sistema son:

- Crear un plan de integración de construcciones que describa las construcciones necesarias en una iteración y los requerimientos de cada construcción.
- Integrar cada construcción antes de que sea sometida a pruebas de integración.

Planificación de una construcción

Se discute como planificar los contenidos de una construcción, independientemente de que partamos de una construcción previa o de no partamos de ninguna. Suponemos que tenemos un número de casos de uso o escenarios y otros requerimientos que han de ser implementados en la iteración actual.

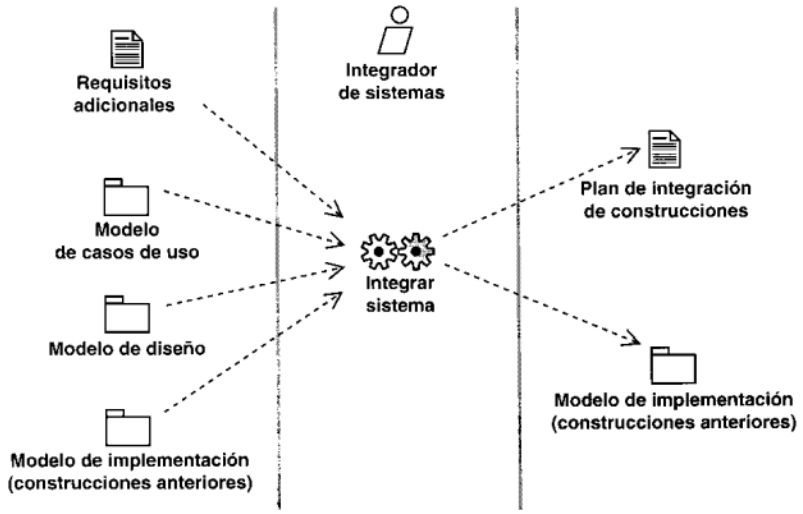
Criterios para crear una construcción:

- Una construcción debería añadir funcionalidad a la construcción previa implementando casos de uso completos o escenarios de éstos.
- Una construcción no debería incluir demasiados componentes nuevos o refinados. Si no es así, puede ser muy difícil integrar la construcción y llevar a cabo las pruebas de integración.
- Una construcción debería estar basada en la construcción anterior, y debería expandirse hacia arriba y hacia los lados en la jerarquía de subsistemas. Esto significa que la construcción inicial debería empezar en capas inferiores; las construcciones subsiguientes se expanden entonces hacia arriba a las capas generales de aplicación. Esto es porque es difícil implementar componentes en las capas superiores antes de que estén colocados y funcionando los componentes necesarias en las capas inferiores.

Los resultados deberían estar recogidos en el plan de integración de la construcción y ser comunicados a los ingenieros de componentes responsables de los subsistemas y componentes de implementación afectados. Los ingenieros de componentes pueden entonces empezar a implementar los requerimientos de los subsistemas y componentes de implementación en la construcción actual y llevar a cabo las pruebas individuales de cada unidad.

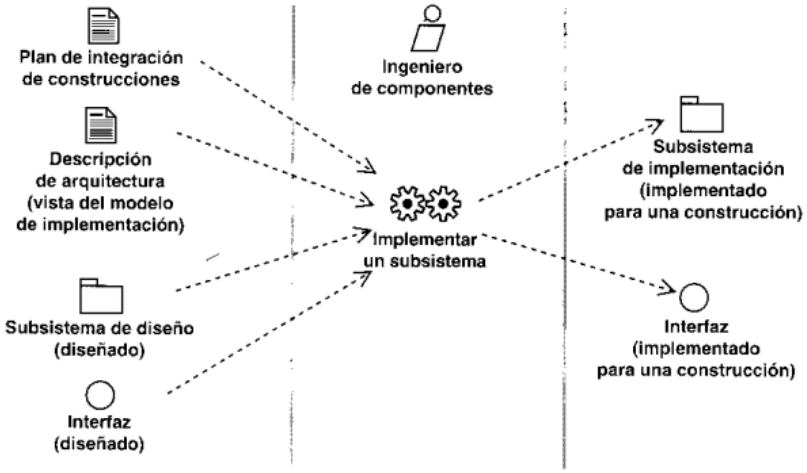
Integración de una construcción

Esto se hace recopilando las versiones correctas de los subsistemas de implementación y de los componentes compilándolos y enlazándolos para generar una construcción.



Implementar un subsistema (por ing. de componentes)

El propósito de implementar un subsistema es el de asegurar que un subsistema cumple su papel en cada construcción, tal y como se especifica en el plan de integración de la construcción. Esto quiere decir que se asegura que los requerimientos implementados en la construcción y aquellos que afectan al subsistema son implementados correctamente por componentes o por otros subsistemas (recursivamente) dentro del subsistema.



Implementar una clase (por ing. de componentes)

El propósito de la implementación de una clase es implementar una clase de diseño en un componente fichero.

Esbozo de los componentes fichero

Es normal implementar varias clases de diseño en un mismo componente fichero. Sin embargo, el tipo de modularización y convenciones de los lenguajes de programación restringen la forma en que los componentes fichero son esbozados.

Generación de código a partir de una clase de diseño

Solo se genera la signatura (la firma) de las operaciones (métodos) las cuales todavía deben ser implementadas.

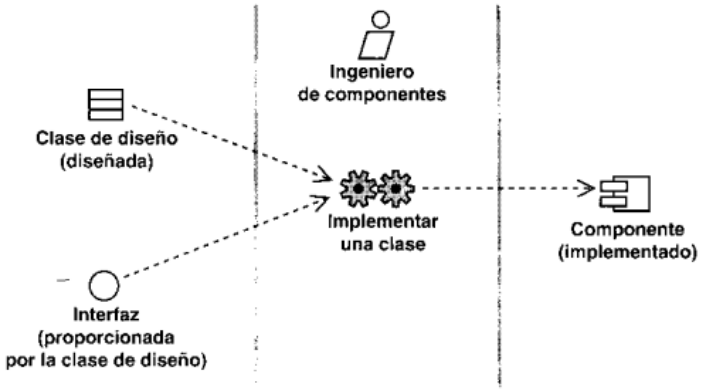
Implementación de operaciones

Cada operación definida por la clase de diseño ha de ser implementada, a no ser que sea virtual (o abstracta) y ésta sea implementada por descendientes de la clase.

La implementación de una operación incluye la elección de un algoritmo y unas estructuras de datos apropiadas, y la codificación de las acciones requeridas por el algoritmo.

Los componentes han de proporcionar las interfaces apropiadas

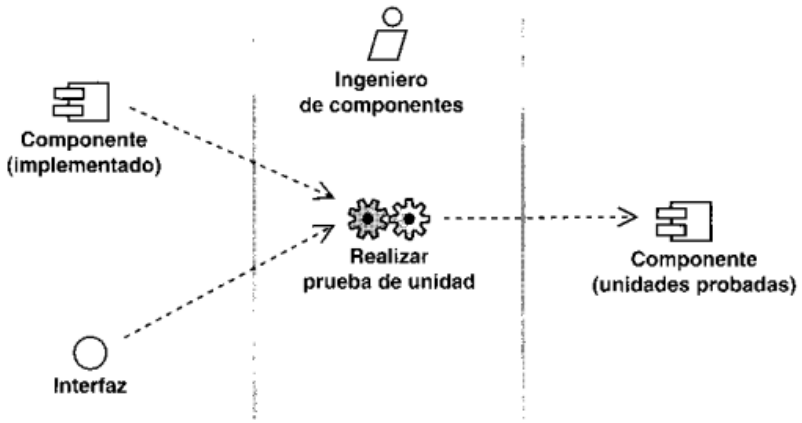
El componente resultante debería proporcionar las mismas interfaces que las clases de diseño que éste implementa.



Realizar prueba de unidad (por ing. de componentes)

Probar los componentes implementados como unidades individuales. Dos tipos de prueba de unidad:

- **Prueba de especificación ("prueba de caja negra"):** se verifica el comportamiento de la unidad observable externamente.
  Se realiza para verificar el comportamiento del componente sin tener en cuenta **cómo** se implementa dicho comportamiento en el componente. Tienen en cuenta la salida que el componente devolverá cuando se le da una determinada entrada en un determinado estado.
- **Prueba de estructura ("prueba de caja blanca"):** verifica la implementación interna de la unidad. *Se fija en el código*.
  Se realizan para verificar que un componente funciona internamente como se quería. El ingeniero de componentes debería también de asegurarse de probar todo el código durante las pruebas de estructura, lo que quiere decir que cada sentencia ha de ser ejecutada al menos una vez. Debería también asegurarse de probar los caminos más importantes en el código.



WORKFLOW DE PRUEBA

Introducción

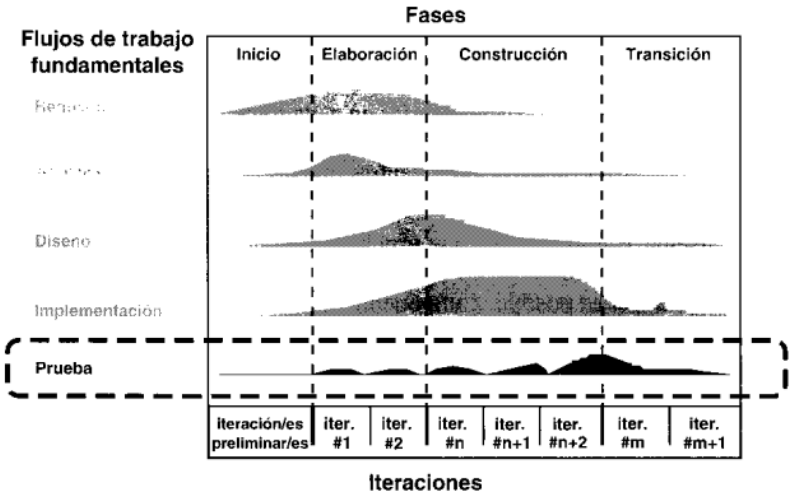
En el flujo de trabajo de la prueba verificamos el resultado de la implementación probando cada construcción, incluyendo tanto construcciones internas como intermedias, así como las versiones finales del sistema a ser entregadas a los clientes.

Propósitos de la prueba

- Planificar las pruebas necesarias en cada iteración, incluyendo las pruebas de integración y las pruebas de sistema. Las pruebas de integración son necesarias para cada construcción dentro de la iteración, mientras que las pruebas de sistema son necesarias sólo al final de la iteración.
- Diseñar e implementar las pruebas creando los casos de prueba que especifican qué probar.
- Realizar las diferentes pruebas y manejar los resultados de cada prueba simultáneamente.

El papel de la prueba en el ciclo de vida del software

La realización de pruebas se centra en las fases de elaboración, cuando se prueba la línea base ejecutable de la arquitectura, y de construcción cuando el grueso del sistema está implementado. Durante la fase de transición el centro se desplaza hacia la corrección de defectos durante los primeros usos y las pruebas de regresión.



Artefactos de la prueba

1. Modelo de pruebas.
2. Caso de prueba.
3. Procedimiento de prueba.
4. Plan de prueba.
5. Defecto.
6. Evaluación de prueba.

Modelo de pruebas

El modelo de pruebas describe principalmente cómo se prueban los componentes ejecutables en el modelo de implementación con pruebas de integración y de sistema. Puede describir también como han de ser probados aspectos específicos del sistema; por ejemplo; si la interfaz de usuario es utilizable y consistente o si el manual de usuario del sistema cumple su cometido. El modelo de pruebas es una colección de casos de prueba, procedimientos de prueba y componentes de prueba.

Caso de prueba

Un caso de prueba especifica una forma de probar el sistema, incluyendo la entrada o resultado con la que se ha de probar y las condiciones bajo las que ha de probarse.

Casos de prueba comunes:

- Un caso de prueba que especifica cómo probar un caso de uso o un escenario específico de un caso de uso (verificación del resultado de la interacción ente actores y el sistema, si se cumplen las precondiciones y postcondiciones y que se siguen la secuencia de acciones especificadas en el caso de uso). Prueba de sistema como **"caja negra"**, es decir, una prueba del comportamiento observable del sistema.
- Un caso de prueba que especifica cómo probar una realización de caso de uso de diseño o un escenario específico de la realización (verificación de la interacción entre componentes que implementan el caso de uso). Prueba de sistema como **"caja blanca"**, es decir, una prueba de la interacción interna entre los componentes del sistema.

Se pueden especificar otros casos de prueba para probar el sistema como un todo. Por ejemplo:

- **Pruebas de instalación:** verifican que el sistema puede ser instalado en la plataforma del cliente y que funcionará correctamente cuando sea instalado.
- **Pruebas de configuración:** verifican que el sistema funciona correctamente en diferentes configuraciones.
- **Pruebas negativas:** intentan provocar que el sistema falle para poder así revelar sus debilidades. Consiste en utilizar el sistema en formas para los que no sido diseñado ("usar mal el sistema a propósito").
- **Pruebas de tensión o de estrés:** identifican problemas con el sistema cuando hay recursos insuficientes o cuando hay competencia por recursos.

Procedimiento de prueba

Un procedimiento de prueba especifica cómo realizar uno a varios casos de pruebas o partes de éstos.

Componente de prueba

Un componente de prueba automatiza uno o varios procedimientos de prueba o partes de ellos.

Se utilizan para probar los componentes en el modelo de implementación, proporcionando entradas de prueba, controlando y monitorizando la ejecución de los componentes a probar y, posiblemente, informando de los resultados de las pruebas.

Plan de prueba

El plan de prueba describe las estrategias, recursos y planificación de la prueba. La estrategia de prueba incluye la definición del tipo de pruebas a realizar para cada iteración y sus objetivos.

Defecto

Un defecto es una anomalía en el sistema.

Evaluación de prueba

Una evaluación de prueba es una evaluación de los resultados de los esfuerzos de prueba.

Trabajadores de la prueba

1. Diseñador de pruebas (ingeniero de prueba).
2. Ingeniero de componentes.
3. Ingeniero de pruebas de integración.
4. Ingeniero de pruebas de sistema.

Diseñador de pruebas

Un diseñador de pruebas es responsable de la integridad del modelo de pruebas, asegurando que el modelo cumple con su propósito. También planean las pruebas, lo que significa que deciden los objetivos apropiados y la planificación de las pruebas. Además, seleccionan y describen los casos de prueba y los procedimientos de prueba correspondientes que se necesitan, y son responsables de la evaluación de pruebas de integración y de sistema cuando éstas se ejecutan.

Ingeniero de componentes

Los ingenieros de componentes son responsables de los componentes de prueba que automatizan algunos procedimientos de prueba.

Ingeniero de pruebas de integración

Los ingenieros de pruebas de integración son los responsables de realizar las pruebas de integración ☺ que se necesitan para cada construcción producida en el flujo de trabajo de la implementación. Las pruebas de integración se realizan para verificar que los componentes integrados en una construcción funcionan correctamente juntos.

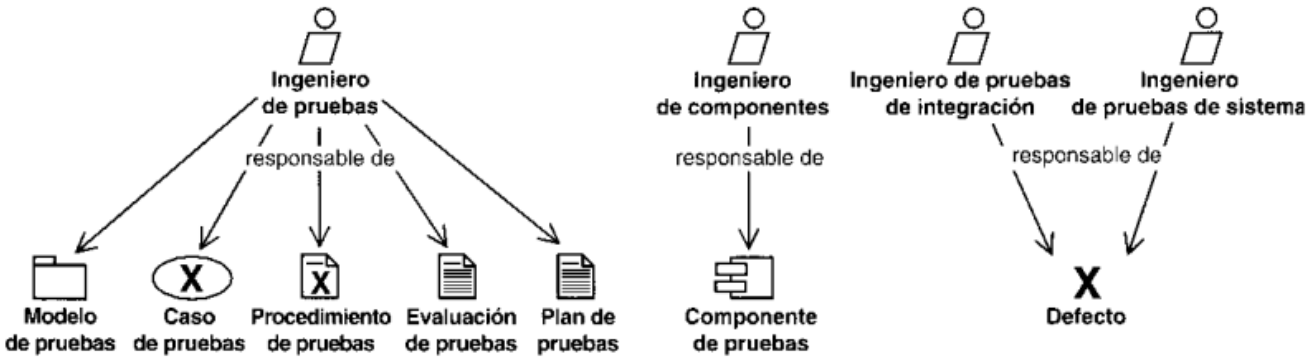
El ingeniero de pruebas de integración se encarga de documentar los defectos en los resultados de las pruebas de integración.

Ingeniero de pruebas de sistema

Un ingeniero de pruebas de sistema es responsable de realizar las pruebas de sistema necesarias sobre una construcción que muestra el resultado (ejecutable) de una iteración completa.

Las pruebas de sistema se llevan a cabo principalmente para verificar las interacciones entre los actores y el sistema, por eso suelen derivarse de los casos de prueba que especifican como probar los casos de uso.

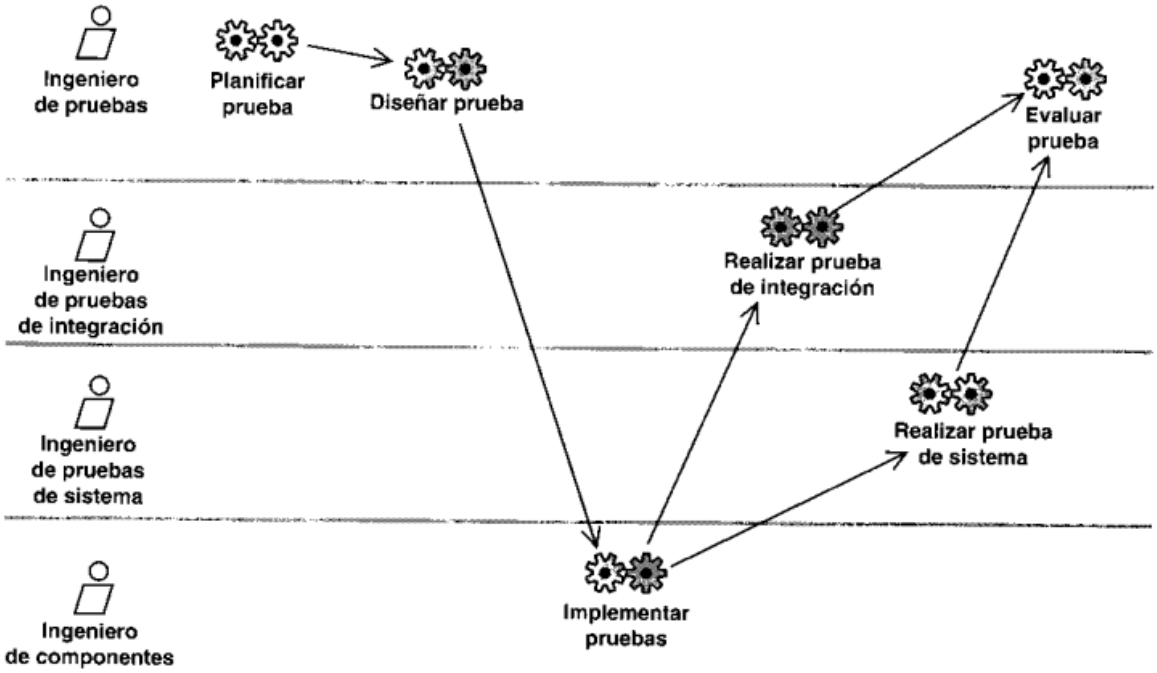
El ingeniero de pruebas de sistema se encarga de documentar los defectos en los resultados de las pruebas de sistema.



Flujo de trabajo (actividades)

1. Los ingenieros de pruebas determinar el objetivo de la prueba a realizar planificando el esfuerzo de prueba a realizar en cada iteración
2. Describen entonces los casos de prueba necesarios y los procedimientos de prueba correspondientes para llevar a cabo las pruebas.
3. Si es posible, los ingenieros de componentes crean a continuación los componentes de prueba para automatizar algunos procedimientos de prueba.
4. Con estos casos, procedimientos y componentes de prueba como entrada, los ingenieros de pruebas de integración y de sistema prueban cada construcción y detectan cualquier defecto que encuentren en ellos.

5. Los defectos sirven como retroalimentación tanto para otros flujos de trabajo, como el diseño y el de implementación, como para los ingenieros de pruebas para que lleven a cabo una evaluación sistemática de los resultados de las pruebas.

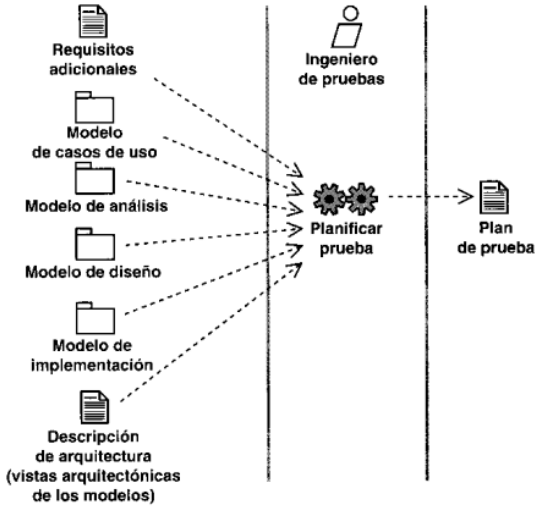


Planificar prueba (ing. de pruebas)

El propósito de la planificación de prueba es planificar los esfuerzos de prueba en una iteración llevando a cabo las siguientes tareas:

- Describiendo una estrategia de prueba.
- Estimando los requerimientos para el esfuerzo de la prueba (ejemplo: recursos humanos y sistemas necesarios).
- Planificando el esfuerzo de la prueba.

En general, se crea el plan de prueba estableciendo las estrategias y recursos.



Diseñar prueba (ing. de pruebas)

Propósitos de diseñar las pruebas:

- Identificar y describir los casos de prueba para cada construcción.
- Identificar y estructurar los procedimientos de prueba especificando cómo realizar los casos de prueba.

Diseño de los casos de prueba de integración

La mayoría de los casos de prueba de integración pueden ser derivados de las realizaciones de casos de uso de diseño, ya que las realizaciones de casos de uso describen cómo interaccionan las clases y los objetos, y por tanto cómo interaccionan los componentes. Se consideran como entrada los diagramas de interacción de las realizaciones de casos de uso.

Diseño de los casos de prueba de sistema

Las pruebas de sistema se usan para probar que el sistema funciona correctamente como un todo. Cada prueba de sistema prueba principalmente combinaciones de casos de uso instanciados bajo condiciones diferentes

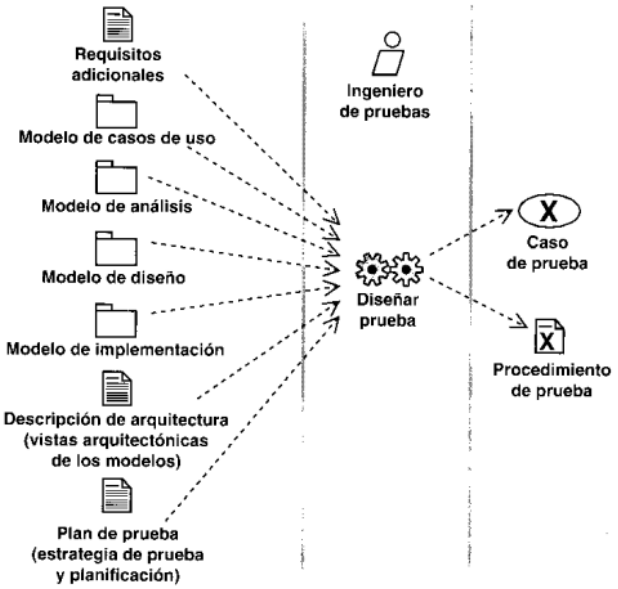
(diferentes configuraciones de hardware, diferentes cargas del sistema, diferentes números de actores y diferentes tamaños de la base de datos).

Diseño de los casos de prueba de regresión

Algunos casos de prueba de construcciones anteriores pueden ser usados para pruebas de regresión en construcciones subsiguientes, aunque no todos los casos de prueba son adecuados para pruebas de regresión.

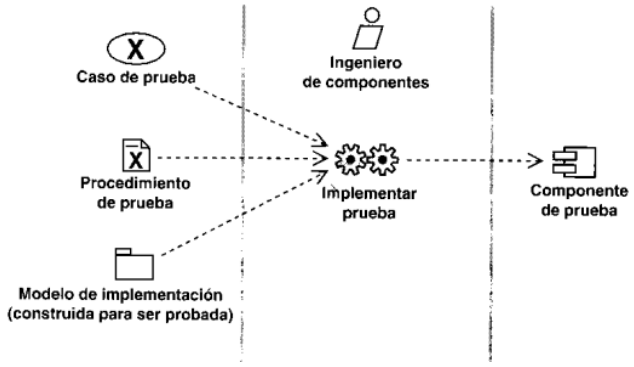
Identificación y estructuración de los procedimientos de prueba

Se intenta reutilizar procedimientos de prueba existentes tanto como sea posible. Los diseñadores de pruebas también intentan crear procedimientos de prueba que puedan ser reutilizados en varios casos de prueba.



Implementar prueba (ing. de componentes)

El propósito de la implementación de las pruebas es automatizar los procedimientos de prueba creando componentes de prueba si esto es posible, ya que no todos los procedimientos de prueba pueden ser automatizados.



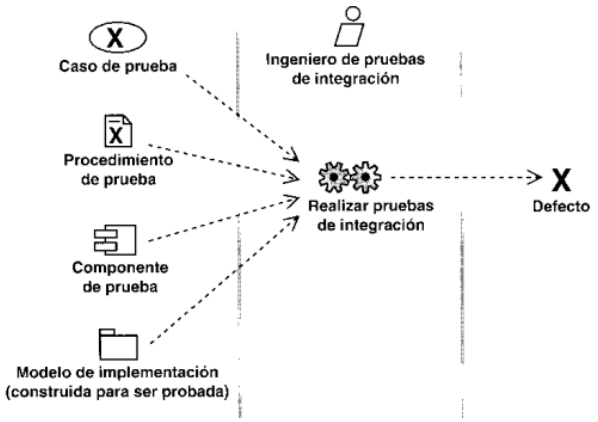
Realizar pruebas de integración (ing. de pruebas de integración)

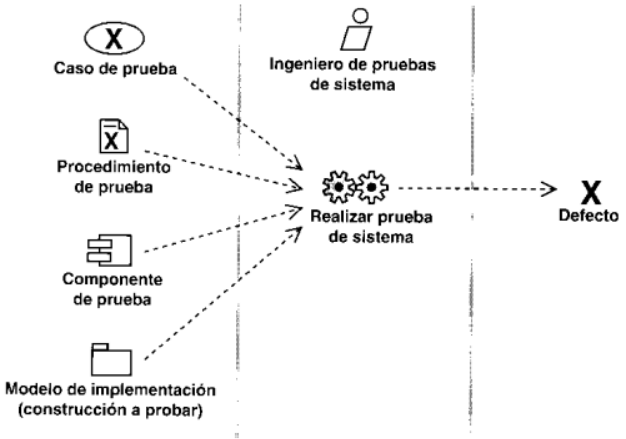
En esta actividad se realizan las pruebas de integración necesarias para cada una de las construcciones creadas en una iteración y se recopilan los resultados de las pruebas.

Realizar prueba de sistema (ing. de pruebas de sistema)

El propósito de la prueba de sistema es el realizar las pruebas de sistema☺.

La prueba de sistema puede empezar cuando las pruebas de integración indican que el sistema satisface los objetivos de calidad de integración fijados en el plan de pruebas de la iteración actual.

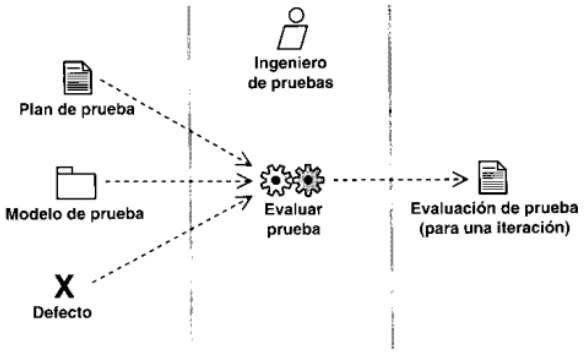




Evaluar la prueba (ing. de pruebas)

El propósito de la evaluación de la prueba es el evaluar los esfuerzos de prueba de una iteración.

Los diseñadores de pruebas evalúan los resultados de la prueba comparando los resultados obtenidos con los objetivos esbozados en el plan de prueba. Éstos preparan métricas que les permiten determinar el nivel de calidad del software y qué cantidad de pruebas es necesario hacer.



PRUEBA

Introducción

Las pruebas intentan demostrar que un programa hace lo que se intenta que haga, así como descubrir defectos en el programa antes de usarlo.

El proceso de prueba tiene dos metas distintas:

1. Demostrar al desarrollador y al cliente que el software cumple con los requerimientos.
2. Encontrar situaciones donde el comportamiento del software sea incorrecto, indeseable o no esté de acuerdo con su especificación.

La primera meta conduce a la prueba de validación; en ella, se espera que el sistema se desempeñe de manera correcta mediante un conjunto dado de casos de prueba, que refleje el uso previsto del sistema. La segunda meta se orienta a pruebas de defectos, donde los casos de prueba se diseñan para presentar los defectos.

Las pruebas no pueden demostrar que el software está exento de defectos o que se comportará como se especifica en cualquier circunstancia.

Las pruebas pueden mostrar sólo la presencia de errores, más no su ausencia.

El propósito de la prueba es ENCONTRAR FALLAS. El testing exitoso es el que encuentro defectos.

Las pruebas pueden dividirse en:

- **Validación:** ¿construimos el producto correcto?
- **Verificación:** ¿construimos bien el producto?

La finalidad de la verificación es comprobar que el software cumpla con su funcionalidad y con los requerimientos no funcionales establecidos. Sin embargo, la validación es un proceso más general. La meta de la validación es garantizar que el software cumpla las expectativas del cliente. La validación es esencial pues, las especificaciones de requerimientos no siempre reflejan los deseos o las necesidades reales de los clientes y usuarios del sistema.

El objetivo final de los procesos de verificación y validación es establecer confianza de que el sistema de software es “adecuado”.

Prueba e inspección

Al igual que las pruebas de software, el proceso de verificación y validación implicaría inspecciones y revisiones de software. Estas últimas analizan y comprueban los requerimientos del sistema, los modelos de diseño, el código fuente, y las pruebas propuestas para el sistema. Éstas son las llamadas técnicas "estáticas".

Técnicas "estáticas": donde no es necesario ejecutar el software para verificarlo.

Las inspecciones se enfocan principalmente en el código del sistema.

Ventajas de la inspección sobre las pruebas

1. Durante las pruebas, los errores pueden enmascarar (ocultar) otras fallas. Cuando un error lleva a salidas inesperadas, nunca se podrá asegurar si las anomalías de salida posteriores se deben a un nuevo error o son defectos colaterales del error original.
2. Las versiones incompletas de un sistema se pueden inspeccionar sin costos adicionales.
3. Una inspección puede considerar también atributos más amplios de calidad de un programa, como el cumplimiento con estándares, la portabilidad y la mantenibilidad. Pueden buscarse inferencias, algoritmos inadecuados y estilos de programación imitados que hagan al sistema difícil de mantener y actualizar.

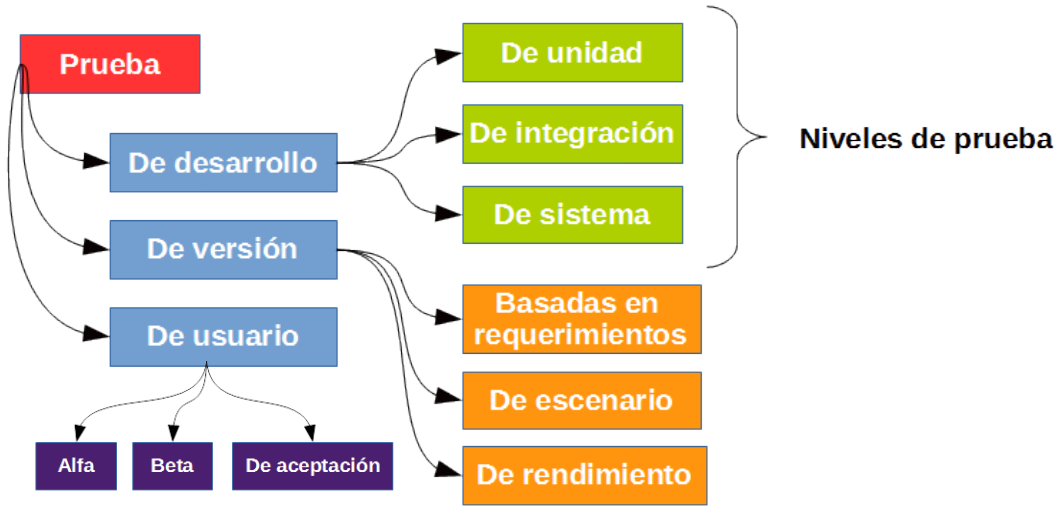
Las inspecciones no sustituyen las pruebas del software, ya que no son eficaces para descubrir defectos que surjan por interacciones inesperadas entre diferentes partes de un programa, problemas de temporización o dificultades con el rendimiento del sistema.

Casos de prueba: son especificaciones de las entradas a las pruebas y la salida esperada del sistema (los resultados de la prueba).

Es posible automatizar la ejecución de pruebas. Los resultados provistos se comparan automáticamente con los resultados establecidos, de manera que no haya necesidad de que un individuo busque errores y anomalías al correr las pruebas.

Etapas de prueba

1. **Pruebas de desarrollo:** donde el sistema se pone a prueba durante el proceso para descubrir errores (*bugs*) y defectos. Intervienen diseñadores y programadores del sistema.
2. **Prueba de versión:** donde un equipo de prueba por separado experimenta una versión completa del sistema, antes de presentarlo a los usuarios.
3. **Pruebas de usuario:** donde los usuarios reales o potenciales del sistema prueban el sistema en su propio entorno.



Pruebas de desarrollo

Las pruebas de desarrollo incluyen todas las actividades de prueba que realiza el equipo que elabora el sistema.

Niveles de prueba

- **Pruebas de unidad:** donde se ponen a prueba unidades de programa, clases de objetos, bloques y paquetes. Las pruebas de unidad deben enfocarse en comprobar la funcionalidad de objetos y métodos.
- **Pruebas de componente (de integración):** se prueban que las unidades trabajan correctamente juntas. Deben enfocarse en probar interfaces de componentes.
- **Pruebas de sistema:** se prueba el sistema completo.

Pruebas de unidad

Es el nivel de prueba más bajo y normalmente lo hace el mismo desarrollador. Involucra clases, bloques, paquetes de servicio.

- **Prueba de caja negra (de especificación):** verifican el comportamiento de la interfaz de la unidad, o sea lo que hace la unidad sin importar como lo hace. Además de verificar si se producen salidas, hay que verificar si la salida es la correcta.
- **Prueba de caja blanca (de estructura):** se verifica si la estructura interna es la correcta. Todos los caminos posibles planteados en el código deben ser contemplados y ejecutados. Por lo general esta prueba se realiza al último.
- **Prueba basada en estados:** prueba la interacción entre las operaciones de una clase, monitoreando los cambios que tienen lugar en los atributos de los objetos. Es importante basarse en diagramas de transición estados, así, al menos cada estado es visitado por lo menos una vez y cada transición es atravesada al menos una vez.

Cuando se pone a prueba las clases de los objetos, hay que diseñar pruebas para brindar cobertura a todas las características del objeto:

- Probar todas las operaciones asociadas al objeto.
- Establecer y verificar el valor de todos los atributos relacionados con el objeto.
- Poner el objeto en todos los estados posibles.

La generalización o herencia provoca que sea más difícil la prueba de las clases de objetos. Por consiguiente, tienen que poner a prueba la operación heredada en todos los contextos en que se utilice.

Siempre que sea posible, se deben automatizar las pruebas de unidad (por ejemplo, JUnit).

Pruebas de componentes (de integración)

Una vez que las unidades han sido certificadas en las pruebas de unidad, estas unidades deberían integrarse en unidades más grandes y finalmente al sistema.

El propósito es determinar si las distintas unidades que han sido desarrolladas trabajan apropiadamente juntas.

En general, los componentes de software son componentes compuestos constituidos por varios objetos en interacción. El acceso a la funcionalidad de dichos objetos es a través de la interfaz de componente definida. Por consiguiente, la prueba de componentes compuestos tiene que enfocarse en mostrar que la interfaz de componente se comporta según su especificación.

Pruebas del sistema

Una vez que se han probado todos los casos de uso por separado se probará el sistema completo.

Las pruebas de sistema durante el desarrollo incluyen la integración de componentes para crear una versión del sistema y, luego, poner a prueba el sistema integrado. Las pruebas de sistema demuestran que los componentes son compatibles, que interactúan correctamente y que transfieren los datos correctos en el momento adecuado a través de sus interfaces.

Cuando se integran componentes para crear un sistema, se obtiene un comportamiento emergente. Esto significa que algunos elementos de funcionalidad del sistema sólo se hacen evidentes cuando se reúnen los componentes. Sin embargo, algunas veces, el comportamiento emergente no está planeado ni se desea. Hay que desarrollar pruebas que demuestren que el sistema sólo hace lo que se supone que debe hacer.

Por lo tanto, las pruebas del sistema deben enfocarse en poner a prueba las interacciones entre los componentes y los objetos que constituyen el sistema.

Pruebas de versión

Las pruebas de versión son el proceso de poner a prueba una versión particular de un sistema que se pretende usar fuera del equipo de desarrollo.

Existen dos distinciones importantes entre las pruebas de versión y las pruebas de sistema durante el desarrollo:

1. Un equipo independiente que no intervino en el desarrollo del sistema debe ser el responsable de las pruebas de versión.
2. Las pruebas del sistema por parte del equipo de desarrollo deben enfocarse en el descubrimiento de bugs en el sistema.

La principal meta del proceso de pruebas de versión es convencer al proveedor del sistema de que éste es suficientemente apto para su uso. Por ello, las pruebas de versión deben mostrar que el sistema entrega su funcionalidad, rendimiento y confiabilidad especificados, y que no falla durante el uso normal.

Las pruebas de versión, por lo regular, son un proceso de prueba de caja negra, donde las pruebas se derivan a partir de la especificación del sistema.

Pruebas basadas en requerimientos

Los requerimientos tienen que escribirse de forma que pueda diseñarse una prueba para dicho requerimiento. Luego, un examinador comprueba que el requerimiento se cumpla. Las pruebas basadas en requerimientos son un enfoque sistemático al diseño de casos de prueba, donde se considera cada requerimiento y se deriva un conjunto de pruebas para éste.

Pruebas de escenario

Es un enfoque donde se crean escenarios típicos de uso y se les utiliza en el desarrollo de casos de prueba para el sistema. Un escenario es una historia que describe una forma en puede usarse el sistema.

Pruebas de rendimiento

Una vez integrado completamente el sistema, es posible probar propiedades emergentes, como el rendimiento y la confiabilidad. Las pruebas de rendimiento deben diseñarse para garantizar que el sistema procese su carga pretendida. Generalmente, esto implica efectuar una serie de pruebas donde se aumenta la carga, hasta que el rendimiento del sistema se vuelve inaceptable.

La prueba de rendimiento significa estresar el sistema al hacer demandas que están fuera de los límites de diseño del software. Esto se conoce como “prueba de esfuerzo”. Este tipo de prueba tiene dos funciones:

1. Prueba el comportamiento de falla del sistema.
2. Fuerza al sistema y puede hacer que salgan a luz defectos que no se descubrirían normalmente.

Las pruebas de esfuerzo son particularmente relevantes para sistemas distribuidos basados en redes de procesadores.

Pruebas de usuario

Las pruebas de usuario o del cliente son una etapa en el proceso de pruebas donde los usuarios o clientes proporcionan entrada y asesoría sobre las pruebas del sistema. Las pruebas de usuario son esenciales, aun cuando se hayan realizado pruebas abarcadoras de sistema y de versión. La razón de esto es que la influencia del entorno de trabajo del usuario tiene un gran efecto sobre la fiabilidad, el rendimiento, el uso y la robustez de un sistema.

Es casi imposible que un desarrollador de sistema replique el entorno de trabajo del sistema, pues las pruebas en el entorno del desarrollador son forzosamente artificiales.

Hay tres tipos de pruebas de usuario:

- **Pruebas alfa:** los usuarios del software trabajan con el equipo de diseño para probar el software en el sitio del desarrollador.
- **Pruebas beta:** una versión del software se pone a disposición de los usuarios, para permitirles experimentar y descubrir problemas que encuentran con los desarrolladores del sistema.
- **Pruebas de aceptación:** donde los clientes prueban un sistema para decidir si está o no listo para ser aceptado por los desarrolladores del sistema y desplegado en el entorno del cliente.

Tipos de prueba

1. **Tests de operación:** el sistema es probado en operación normal. Mide la confiabilidad del sistema y se pueden obtener mediciones estadísticas.
2. **Tests de escala completa:** ejecutamos el sistema al máximo, todos los parámetros se enfocan en valores máximos, todos los equipos conectados, usados por muchos usuarios ejecutando casos de uso simultáneamente.
3. **Tests de performance o capacidad:** el objetivo es medir la capacidad de procesamiento del sistema. Los valores obtenidos son comparados con los requeridos.
4. **Tests de sobrecarga (de estrés):** determinan cómo se comporta el sistema cuando es sobrecargado. No se puede esperar que supere esta prueba, pero sí que no se venga abajo, que no ocurra una catástrofe. Cuántas veces se cayó el sistema es una medida interesante. Es importante ver como “queda” el sistema.
5. **Tests de requerimientos:** estos tests se mapean (rastrean) directamente desde la especificación de requerimientos.
6. **Tests de documentación:** se prueba la documentación del sistema.
7. **Tests de aceptación:** ejecutado por la organización que solicita el sistema. Genera la aceptación o no del sistema.
8. **Tests de instalación:** se verifica que el sistema pueda ser instalado en la plataforma del cliente y que el sistema funcionará correctamente cuando sea instalado.
9. **Tests de configuración:** verifica que el sistema funciona correctamente en diferentes configuraciones.
10. **Tests negativo:** el sistema se usa en forma incorrecta intencionalmente para probar casos especiales.
11. **Test de procesos de negocio**
12. **Test de seguridad**

Desarrollo dirigido por pruebas (TDD)

El desarrollo dirigido por pruebas (TDD, *Test-Driven Development*) consiste en desarrollar el código incrementalmente, junto con una prueba para ese incremento. No se avanza hacia el siguiente incremento sino hasta que el código diseñado para la prueba.

WORKFLOW DE DESPLIEGUE

Propósito

El propósito del despliegue es volcar el producto finalizado a sus usuarios; presentar el software. Incluye los siguientes tipos de actividades:

- Probar el software en el ambiente donde va a operar.
- Empaquetar el software.
- Distribuir el software.
- Instalar el software.
- Capacitar a los usuarios finales.
- Migrar el software existente o convertir bases de datos.

El centro de atención del despliegue se centra en la fase de transición. Durante la fase de transición, el usuario final puede probar el sistema para saber cómo trabaja en un ambiente real de trabajo.

Trabajadores del despliegue

1. Gerente de despliegue.
2. Gerente de proyecto.
3. Escritor técnico.
4. Desarrollador de cursos.
5. Artista gráfico.
6. Tester.
7. Implementador.

Gerente de despliegue

Planifica y organiza el despliegue. Es responsable de la retroalimentación de las beta test asegurando que el producto es empaquetado apropiadamente para su envío.

Gerente de proyecto

Es el intermediario con los clientes. Es responsable de aprobar el despliegue según la retroalimentación y las evaluaciones de resultados de las pruebas; y de la aceptación del cliente sobre el envío del producto.

Escritor técnico

Planea y produce material de soporte para los usuarios finales.

Desarrollador de cursos

Planea y produce material para capacitación.

Artista gráfico

Es el responsable de todo el material artístico del producto.

Tester

Ejecuta pruebas de aceptación y es responsable de asegurar que el producto se ha probado adecuadamente.

Implementador

Crea los scripts de instalación y los artefactos relacionados que ayudaran al usuario final a instalar el producto.

Artefactos del despliegue

El artefacto principal es la *release*, que consiste en:

- El **software ejecutable**, en todos los casos.
- **Artefactos de instalación**: scripts, herramientas, archivos, guías, información de licencias.
- **Notas de la release**: notas que describen la release para el usuario final.
- **Material de soporte:** manual de usuario, manual de operaciones y mantenimiento.
- **Materiales de capacitación**

En el caso de productos manufacturados, se necesitan algunos artefactos adicionales, como:

- La lista de materiales incluida en el producto.
- Release master
- Paquete (contenedor del producto) y arte del producto.
- Medio de almacenamiento (CD, DVD, etc.)

Otros artefactos usados pero no son necesariamente entregados al cliente son:

- Los resultados de las pruebas.
- Resultados de retroalimentación.
- Resumen de las pruebas de evaluación.

Flujo de trabajo (actividades)

1. Plan de despliegue.
2. Desarrollar material de soporte.
3. Probar el producto en el lugar de desarrollo.
4. Crear la release (producto final).
5. Beta test de release.
6. Probar el producto en el lugar de instalación.
7. Empaquetar el producto (opc).
8. Proveer acceso a sitio de descarga (opc.)

Plan de despliegue

Dado que un despliegue exitoso está definido por la voluntad del cliente de usar el nuevo software, el planeamiento del despliegue no sólo debe tener en cuenta cómo y cuándo entregar el nuevo software, sino también debe asegurarse que el usuario final tiene toda la información necesaria para recibir el nuevo software adecuadamente y comenzar a usarlo. Para asegurarse de eso, los planes de despliegue incluyen una prueba beta del programa, para ir asesorando al usuario de forma temprana a través de las betas que todavía están en construcción. El planeamiento de despliegue general del sistema requiere un alto grado de colaboración y preparación del cliente. Una conclusión exitosa del proyecto del software puede ser impactada severamente por factores fuera del desarrollo mismo del software, tales como el edificio donde se instalará el software, la infraestructura de hardware fuera de lugar y usuarios que no están lo suficientemente preparados para adaptarse al nuevo software.

Desarrollar material de soporte

El material de soporte cubre toda la información que será requerida por el usuario final para instalar, operar, usar y mantener el sistema. También incluye material de capacitación para que todos los usos posibles que se le pueda dar al sistema sean efectivos.

Probar el producto en el lugar de desarrollo

Probar el producto en el lugar de desarrollo determina si el producto tiene la madurez suficiente para ser entregado como producto final o como distribución para beta-testers. El testeo de betas se hace para pulir un amplio rango de

detalles, permitiendo al usuario final retroalimentar y mejorar el producto antes de su entrega final y en el caso de sistemas desarrollados a medida, la prueba beta puede ser una instalación piloto en el lugar donde se instalará el sistema.

Crear la release

Hay que asegurarse que el producto está preparado para la entrega al cliente. La release (producto final) consiste en todo lo que el usuario final necesitará para instalar y ejecutar el software finalmente.

Beta test de release

Esto requiere que el producto sea instalado por el cliente, quien proveerá información sobre su performance y usabilidad. En el contexto del desarrollo iterativo, el beta testing es esencial para asegurarse que las expectativas del cliente fueron satisfechas y la información provista por el usuario, se retroalimente a la próxima iteración del desarrollo.

Probar el producto en el lugar de instalación

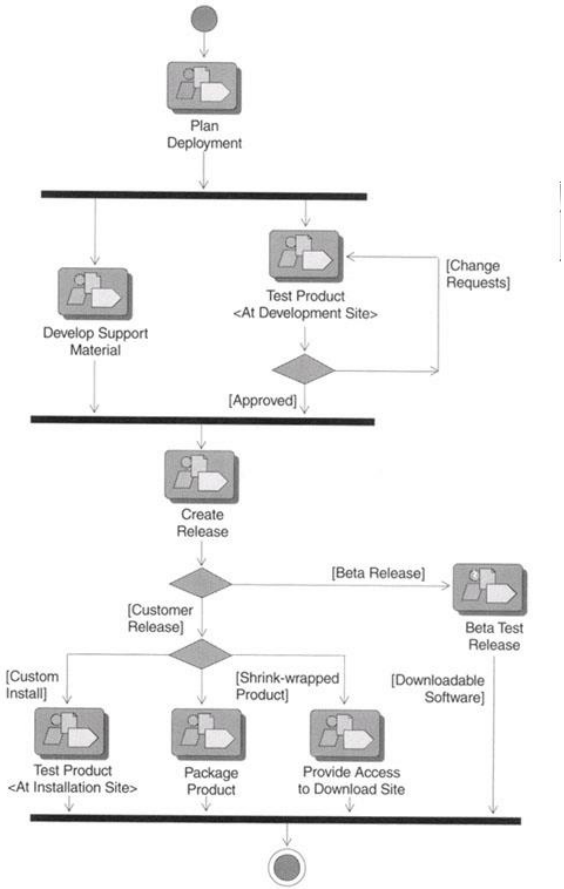
El producto debe ser instalado y probado por el cliente. Basándonos en las iteraciones y pruebas anteriores, en esta prueba no deberíamos encontrarnos con sorpresas y debería ser solo una formalidad para que el cliente acepte el sistema.

Empaquetar el producto (opc.)

Estas actividades opcionales describen lo que debe pasar para producir un producto de "software empaquetado". En este caso, el producto final se guarda como producto maestro para su producción en masa y luego empaquetado en cajas con la lista de contenidos para su envío al cliente.

Proveer acceso a sitio de descarga (opc.)

El producto es accesible a través de internet para su descarga.



EVOLUCIÓN DEL SOFTWARE

Introducción

El desarrollo del software no se detiene cuando un sistema se entrega, sino que continúa a lo largo de la vida de éste. Después de distribuir un sistema, inevitablemente debe modificarse, con la finalidad de mantenerlo útil. Tanto los cambios empresariales como las expectativas del usuario generan nuevos requerimientos para el software existente.

La evolución de un sistema rara vez puede considerarse en aislamiento. Los cambios al entorno conducen a cambios en el sistema que, a la vez, pueden generar más cambios en el entorno.

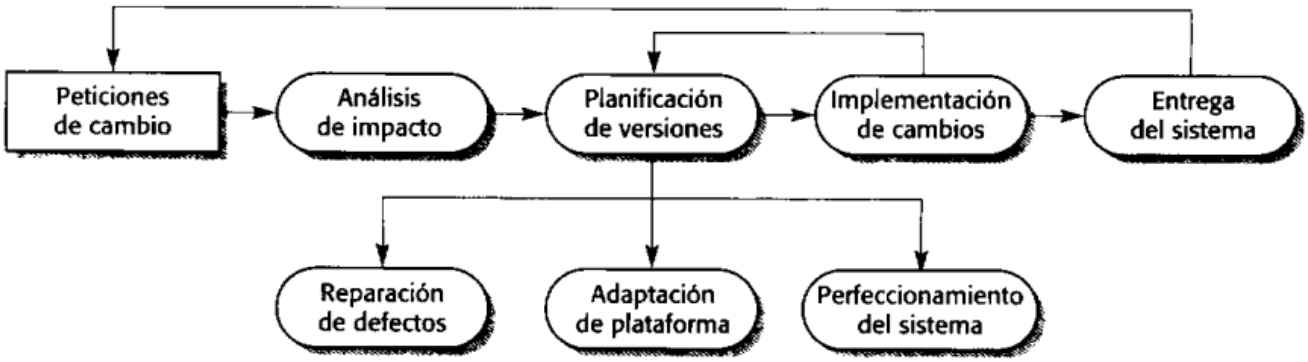
La ingeniería de software se debe considerar como un proceso en espirar, con requerimientos, diseño, implementación y pruebas continuas, a lo largo de la vida del sistema. Esto comienza por crear la versión 1 del sistema. Una vez entregada, se proponen cambios y casi de inmediato comienza el desarrollo de la versión 2.

Mantenimiento de software: es el proceso de cambiar el software después de la entrega. Incluye actividades de proceso adicionales como la comprensión del programa, además de las actividades normales del desarrollo del software.

Procesos de evolución

Los procesos de evolución del software varían dependiendo del tipo de software que se mantiene, de los procesos de desarrollo usados en la organización y de las habilidades de las personas que intervienen. En algunas organizaciones, la evolución es un proceso informal, donde las solicitudes de cambio provienen de conversaciones entre los usuarios del sistema y los desarrolladores. En otras compañías, se trata de un proceso formalizado con documentación estructurada generada en cada etapa del proceso.

Las propuestas de cambio al sistema son el motor para la evolución del sistema en todas las organizaciones. Estos cambios provienen de requerimientos existentes que no se hayan implementado en el sistema liberado, de peticiones de nuevos requerimientos, de reportes de bugs de los participantes del sistema, y de nuevas ideas para la mejora del software por parte del equipo de desarrollo del sistema. Los procesos de identificación de cambios y evolución del sistema son cíclicos y continúan a los largo de la vida de un sistema.

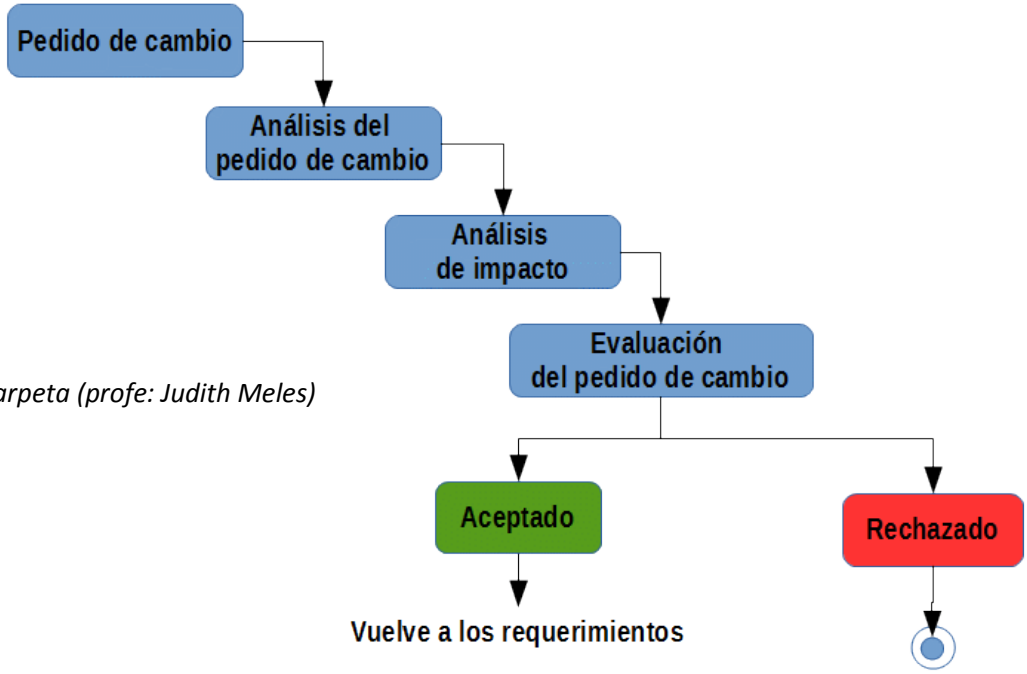


El proceso de evolución, el cual incluye actividades fundamentales de análisis de cambio, planeación de la versión, implementación del sistema y su liberación a los clientes. El costo y el impacto de dichos cambios se valoran para saber que tanto resultará afectado el sistema por el cambio y cuánto costaría implementarlo. Si los cambios propuestos se aceptan, se planea una nueva versión del sistema. Durante la planeación de la versión se consideran todos los cambios propuestos (reparación de fallas, adaptación y nueva funcionalidad). Entonces se toma una decisión acerca de cuáles cambios implementar en la siguiente versión del sistema. Después de implementarse, se

valida y libera una nueva versión del sistema. Luego, el proceso se repite con un conjunto nuevo de cambios propuestos para la siguiente liberación.

La primera etapa de implementación del cambio puede involucrar la comprensión del programa, sobre todo si los desarrolladores del sistema original no son los responsables de implementar el cambio.

En carpeta (profe: Judith Meles)



Mantenimiento

El mantenimiento del software es el proceso general de cambiar un sistema después de que éste se entregó. Los cambios realizados al software van desde los simples para corregir errores de codificación, los más extensos ara corregir errores de diseño, hasta mejorías significativas para corregir errores de especificación o incorporar nuevos requerimientos.

Tipos de mantenimiento de software

- **Reparaciones de fallas ("mantenimiento correctivo"):** los errores de codificación por lo general son relativamente baratos de corregir; los errores de diseño son más costosos, ya que pueden implicar la reescritura de muchos componentes. Los errores de requerimientos son los más costosos de reparar debido a que podría ser necesario un extenso rediseño del sistema.
- **Adaptación ambiental ("mantenimiento adaptativo"):** este tipo de mantenimiento se requiere cuando algún aspecto del entorno del sistema, como el hardware, la plataforma operativa del sistema u otro soporte, cambia el software.
- **Adición de funcionalidad ("mantenimiento perfectivo"):** este tipo de mantenimiento es necesario cuando varían los requerimientos del sistema, en respuesta a un cambio organizacional o empresarial.

Predicción de mantenimiento

Se debe tratar de predecir qué cambios deben proponerse al sistema y qué partes del sistema es probable que sean las más difíciles de mantener. También hay que tratar de estimar los costos de mantenimiento globales para un sistema durante cierto lapso de tiempo.

Predecir el número de peticiones de cambio para un sistema requiere un entendimiento de la relación entre el sistema y su ambiente externo. Para evaluar las relaciones entre un sistema y su ambiente, se debe valorar:

1. El número y la complejidad de las interfaces del sistema: cuantas más interfaces y más complejas sean, más probable será que se requieran cambios de interfaz conforme se propongan nuevos requerimientos.

2. El número de requerimientos de sistema inherentemente inestables: los requerimientos que reflejan políticas y procedimientos de la organización son más inestables que los requerimientos que se basan en características de un dominio estable.
3. Los procesos de negocio donde se usa el sistema: a medida que evolucionan los procesos de negocio, generan peticiones de cambio del sistema.

Reingeniería de software

Significa volver a hacer ingeniería. Aplicado a sistemas heredados. Se crea un nuevo sistema a partir de uno ya existente.

Para hacer que los sistemas heredados sean más sencillos de mantener, se pueden someter a reingeniería para mejorar su estructura y entendimiento. La reingeniería puede implicar volver a documentar el sistema, refactorizar su arquitectura, traducir los programas a un lenguaje de programación moderno, y modificar y actualizar la estructura y los valores de los datos del sistema. La funcionalidad del software no cambia y, normalmente, conviene tratar de evitar grandes cambios en la arquitectura del sistema.

Ventajas de la reingeniería frente a la sustitución

1. **Reducción del riesgo:** pueden cometerse errores en la especificación del sistema o tal vez haya problemas de desarrollo, las demoras en la introducción del nuevo software podrían significar que la empresa está perdida y que se incurrirá en costos adicionales.
2. **Reducción de costos:** el costo de la reingeniería puede ser significativamente menor que el costo de desarrollar software nuevo.

Actividades de la reingeniería

1. **Traducción del código fuente:** el programa se convierte de un lenguaje de programación antiguo a una versión más moderna, o a un lenguaje diferente.
2. **Ingeniería inversa:** el programa se analiza y se extrae información del él. Por lo general es un proceso automatizado.
3. **Mejoramiento de la estructura del programa:** la estructura de control del programa se analiza y modifica para facilitar su lectura y comprensión. Puede ser automatizado con alguna intervención manual.
4. **Modularización del programa:** las partes relacionadas del programa se agrupan y, donde es adecuado, se elimina la redundancia. En algunos casos, implica refactorización arquitectónica. Es un proceso manual.
5. **Reingeniería de datos:** los datos procesados por el programa cambian para reflejar cambios al programa. Significa la redefinición de los esquemas de bases de datos y convertir las bases de datos existentes a la nueva estructura.

Cuando se aplica la reingeniería no necesariamente se requiere de todos las actividades.

Desventajas

- No es posible convertir un sistema funcional a un sistema orientado a objetos.
- Grandes cambios no pueden hacerse de forma automatizada.
- Los sistemas con reingeniería probablemente no serán tan mantenibles como los sistemas nuevos desarrollados con técnicas modernas.

Refactorización

La refactorización es el proceso de hacer mejoras a un programa para frenar la degradación mediante el cambio. Estos significa modificar un programa para mejorar su estructura, reducir su complejidad o hacerlo más fácil de entender.

Mientras se refactorice un programa, no se debe agregar funcionalidad, sino que hay que concentrarse en la mejora del programa.

Aunque la reingeniería y la refactorización tienen la intención de hacer el software más fácil de entender y cambiar, no son lo mismo.

Refactorización vs reingeniería

La reingeniería se lleva a cabo después de haber mantenido un sistema durante cierto tiempo. Se crea un nuevo sistema a partir de un sistema heredado utilizando herramientas automatizadas.

La refactorización es un proceso continuo de mejoramiento debido al proceso de desarrollo y evolución. Tiene la intención de evitar la degradación de la estructura y el código que aumentan los costos y las dificultades por mantener un sistema.

Mejoras de la refactorización

La refactorización mejora:

1. **Código duplicado:** el mismo código se incluye en diferentes partes del programa. Éste se descarta o se implementa como un solo método.
2. **Métodos largos:** si hay métodos largos, deben rediseñarse en varios métodos más pequeños.
3. **Sentencias *case* (*switch*):** en un lenguaje orientado a objetos se pueden reemplazar mediante polimorfismo.
4. **Aglomeración de datos:** las aglomeraciones de datos ocurren cuando el mismo grupo de objetos de datos vuelven a ocurrir en muchos lugares en un programa. Pueden sustituirse con un objeto que encapsule todos los datos.
5. **Generalidad especulativa:** esto ocurre cuando los desarrolladores incluyen generalidad en un programa, en caso de que se requiera en el futuro. Por lo general, eso simplemente puede eliminarse.

Rediseño arquitectónico

Se realizan cambios en la arquitectura del sistema.

Cómo decidir

- Baja calidad y bajo valor de negocio → desechar.
- Baja calidad y alto valor de negocio → Reingeniería.
- Alta calidad y bajo valor de negocio → Si el cambio es caro, desechar.
- Alta calidad y alto valor de negocio → Mantenimiento normal.

CONTENIDO