UNIDAD 1

UML 2.0.

Es un lenguaje que sirve para visualizar, documentar, especificar y comunicar.

¿Por qué es un lenguaje?
Porque al igual que un lenguaje cualquiera, tiene símbolos específicos y todos ellos tienen un significado dentro de un contexto determinado. También es algo universal, es decir, cualquiera que se encuentre dentro de este contexto lo puede entender y comprender.
Los diagramas, vistas y símbolos tienen el mismo significado en cualquier parte del mundo y se usan para modelar algo particular, por eso es que se trata de un lenguaje, porque define reglas y normas para utilizar sus elementos.

Diagramas: Representación gráfica de un conjunto de elementos, visualizado la mayoría de las veces como un grafo conexo de nodos y arcos. Los diagramas se dibujan para visualizar un sistema desde diferentes perspectivas, de forma que un diagrama es una proyección de un sistema.

Tipos de diagramas:
- De Clases: muestran un conjunto de clases, interfaces y colaboraciones, así como sus relaciones. Modelan Estructura.
- De Objetos: muestra un conjunto de objetos y sus relaciones. Representan instantáneas estáticas de las instancias de una clase. Modelan Estructura.
- De Componentes: representan la encapsulación de una clase, junto con sus interfaces, puertos y estructura interna, la cual está formada por otros componentes anidados y conectores. Modelan Estructura.
- De Casos de Uso: muestran un conjunto de casos de uso, actores y sus relaciones. Organizan y modelan el Comportamiento del sistema.
- De Interacción: muestra una interacción, que consta de un conjunto de objetos y roles, incluyendo los mensajes que pueden ser enviados entre ellos. Modelan el Comportamiento del sistema. Se pueden usar para modelar el flujo de control particular de un caso de uso. Cubre la vista dinámica del sistema.
    - De Comunicación: resalta la organización estructural de los objetos y roles que envían y reciben mensajes.
    - De Secuencia: resalta la ordenación temporal de los mensajes.
    - De Tiempos: muestra los tiempos reales entre diferentes objetos o roles, en oposición a la simple secuencia relativa de mensajes. Modelan Comportamiento.
- De Estados: muestra una máquina de estados, que consta de estados, transiciones, eventos y actividades. Modelan Comportamiento.
- De Actividades: muestra la estructura de un proceso como el flujo de control y datos paso a paso. Modelan Comportamiento.
- De Despliegue: muestra la configuración de nodos de procesamiento en tiempo de ejecución y los artefactos que residen en ellos. Modelan Estructura.
- De Artefactos: muestra los constituyentes físicos de un sistema en el computador. Modelan Estructura.
- De Paquetes: muestra la descomposición del propio modelo en unidades organizativas y sus dependencias. Modelan Estructura.

Vistas de UML
1. *De Casos de Uso*: Muestra los casos de uso que describen el comportamiento del sistema tal y como es percibido por los usuarios finales, analistas y testers. Los aspectos estáticos de esta vista se capturan con el diagrama de casos de uso, y los dinámicos con los diagramas de interacción, de estados y de actividades. Comportamiento.
2. *De Despliegue*: Contiene los nodos que conforman la topología de hardware sobre la que se ejecuta el sistema. Los aspectos estáticos de esta vista se capturan con el diagrama de despliegue, y los dinámicos con los diagramas de interacción, de estados y de actividades. Topología de hardware.

3. *De Diseño*: Comprende las clases, interfaces y colaboraciones que forman el vocabulario del problema y la solución. Soporta principalmente los requerimientos funcionales del sistema. Los aspectos estáticos se capturan con el diagrama de clases y de objetos, y los dinámicos con los diagramas de interacción, de estados y de actividades. Vocabulario.
4. *De Implementación*: Comprende los artefactos que se utilizan para ensamblar y poner en ejecución el sistema físico. Los aspectos estáticos se capturan con el diagrama de artefactos, y los dinámicos con los diagramas de interacción, de estados y de actividades. Ensamblado.
5. *De Procesos*: Muestra el flujo de control entre las diversas partes del sistema, incluyendo mecanismos de concurrencia y sincronización. Los aspectos estáticos se capturan con el diagrama de clases y de objetos, y los dinámicos con los diagramas de interacción, de estados y de actividades., pero haciendo énfasis en las clases activas que controlan el sistema y los mensajes que fluyen entre ellas. Flujo de control.

También existen otras vistas como la vista de seguridad o de datos.

PATRONES GRASP

Son importantes porque ayudan a construir clases más cohesivas y el menor grado de dependencia hacia otras clases (acoplamiento), por esto es que existen estos patrones para solucionar el problema de qué responsabilidades asignarle a una clase determinada. (Responsabilidades 'hacer' y 'conocer').

- **Creador**: la clase A debe tener la responsabilidad de crear a la clase B si:
    - A contiene a B
    - A tiene la información necesaria para crear a B
    - A conoce a B

  Ejemplo: Si hay que crear un pedido con sus respectivos detalles de pedido, la clase responsable de crear el pedido es muy probable que sea el cliente, y el responsable de crear los detalles de pedido será el propio pedido que está formado por esos detalles de pedido.
- **Experto en Información**: la responsabilidad debe ser asignada a la clase que tiene la información necesaria para realizar esa acción.

  Ejemplo: Para calcular el total de un pedido, la clase que tiene la información necesaria es la clase pedido, para eso necesita los subtotales que los tienen los detalles de pedido, entonces la clase pedido les pide a los detalles de pedido que contiene, que calculen sus respectivos subtotales, para que luego la clase pedido pueda calcular el total del pedido sumando los subtotales.
- **Alta cohesión**: la responsabilidad a una clase debe asignarse de tal manera que se mantenga alta la cohesión

  Ejemplo: Tengo una clase película y una clase gestor, y quiero averiguar si una película ya se estrenó o no. Una solución es que el gestor le pida la fecha de estreno a la película y que el gestor verifique si la película ya se estrenó, pero de esta manera no se mantiene alta la cohesión porque le estás asignando una responsabilidad al gestor que tranquilamente la podría hacer la clase película, ya que tiene la información para hacerlo. La solución más adecuada sería que sea la propia película quien tenga la responsabilidad de verificar si ya fue estrenada o no, y le devuelva true o false al gestor.
- **Bajo acoplamiento**: la responsabilidad a una clase debe asignarse de tal manera que se mantenga bajo el acoplamiento (no crear dependencias que sean innecesarias).

  Ejemplo: Tengo una clase torneo y una clase fecha de torneo. ¿Quién sería más conveniente que tenga la responsabilidad de crear la fecha de torneo, el gestor o el torneo? La solución más adecuada es que sea el torneo el encargado de crear la fecha del mismo, ya que contiene la información necesaria para poder hacerlo, además de que seguramente ya existe una relación estructural entre esas dos clases (cosa que con el gestor no pasa, y esto aumentaría el acoplamiento).
- **Controlador**: qué clase debe ser responsable de entender un evento del sistema (provocado por un actor externo). Para eso se crea una clase controlador que se encarga de manejar o controlar las distintas actividades que se deben realizar en un escenario de un determinado caso de uso. Este patrón dice que la clase controlador deberá tener la responsabilidad de coordinar el caso de uso y es quien efectivamente sabe cuáles son los pasos a seguir, y lo que hace es delegar a los distintos objetos el trabajo que se tiene que hacer. Además, desacopla la interfaz de las clases de entidad. Existen dos tipos:

- El de fachada: representa el sistema global, dispositivo o subsistema, y se utiliza cuando no existen demasiados eventos del sistema o cuando no hay posibilidad de que la interfaz redireccione los distintos mensajes de los eventos a controladores alternativos.
- Controlador de casos de uso: existe un controlador diferente para cada caso de uso, aquí el controlador es una clase de fabricación pura que se utiliza para dar soporte al sistema.

Sin la clase controlador, es la interfaz quien debe conectarse con las clases de entidad y esa no es su responsabilidad. La responsabilidad de la interfaz es vincularse con el ambiente, No debería de tener metida la lógica de negocio (no debería saber cuál es el paso siguiente, sólo debería recibir la información del ambiente y pasarla al controlador, y que sea él quien tenga la responsabilidad de saber qué es lo que sigue).

WF DE ANÁLISIS

Tiene el protagonismo en las últimas iteraciones de la fase de inicio y en las primeras de la fase de elaboración.

De este wf sale el *modelo de análisis*, que es una primera aproximación al diseño, está escrito en el lenguaje del desarrollador y permite ver el sistema sin tener los detalles de la implementación, lo cual hace que sea mucho más fácil entender el sistema y tener un primer pantallazo de los que hace el mismo.

A diferencia del modelo de diseño, el modelo de análisis no resuelve todos los requerimientos y es genérico con respecto a la implementación.

En el wf de requerimientos uno determinaba los requerimientos, pero en el análisis es donde se especifican esos requerimientos y empezamos a abrir los casos de uso y no solamente ver lo que hacen, sino también ver cómo lo hacen (empezamos a pensar cómo vamos a programar esos casos de uso, pero sin especificar un lenguaje de programación; pensamos en una solución genérica que luego será especificada en el diseño).

La entrada del wf de análisis es el modelo de requerimientos (los casos de uso que se descubrieron).

Los trabajadores de wf de análisis son:

- Arquitecto: se encarga de mantener consistente y legible como un todo el modelo de análisis, también se encarga de refinar los paquetes y las clases que participan en las diferentes realizaciones de casos de uso-análisis.
- Ingeniero de casos de uso: se encarga de construir las realizaciones de casos de uso-análisis, respectando las clases participantes.
- Ingeniero de componentes: se encarga de mantener consistentes los paquetes y las clases que participan en los distintos casos de uso-análisis.

UNIDAD 2

WF DE DISEÑO

Sus salidas son el modelo de diseño y el modelo de despliegue; y tiene como entradas los requerimientos del wf de requerimientos y el modelo de análisis del wf de análisis.

En el diseño construimos un modelo físico que ya no es genérico con respecto a la implementación, sino que es específico.

Los trabajadores siguen siendo los mismos que en el análisis. Acá resolvemos los requerimientos no funcionales, aunque podemos posponer algunos para el momento de la implementación.

Acá se diseña la arquitectura y es por este motivo que este modelo debe ser mantenido a lo largo de todo el ciclo de vida del producto, a diferencia del análisis.

No modelamos la solución en términos lógicos, sino en términos físicos; se ocupa de la solución de todos los aspectos que tienen que ver con la implementación.

Tiene un papel preponderante en las últimas iteraciones de la fase de elaboración y en las primeras iteraciones de la fase de construcción.

Modelo de Diseño: Es un modelo de objetos que describe la realización física de los casos de uso, centrando la atención en como los RF y los RNF, junto con otras restricciones de implementación, tienen impacto en el sistema a considerar

Modelo de Despliegue: Es un modelo de objetos que describe la distribución física del sistema en cuanto a la distribución de la funcionalidad en los nodos de cómputo. Puede describir diferentes configuraciones de red. Representa una correspondencia entre la arquitectura del sw y la arquitectura del hw. Es la descripción de la arquitectura.

PRINCIPIOS DE DISEÑO

Características del buen diseño

- Identificación y tratamiento de excepciones: debemos permitirle al usuario que, frente a una situación de error, o acciones inadecuadas, o una excepción, él sepa cómo tiene que seguir a partir de ahí, de tal manera que pueda recuperarse del mismo.
- Prevención y tolerancia de defectos: debemos realizar un software que contemple si algo sale de lo normal.
- Independencia de componentes: se trata que los componentes sean independientes unos de otros. Se busca obtener un bajo acoplamiento y una alta cohesión.

PRINCIPIOS DE DISEÑO

Son una guía de las buenas prácticas que se deben seguir en el diseño de software, no siempre se pueden cumplir todos, pero la idea de estos principios es que sirvan como guía para generar un software mucho más mantenible, extensible y flexible.

Es una técnica que se aplica para diseñar o construir software, para hacerlo más flexible, extensible y mantenible.

Son guías de alto nivel que se aplican a cualquier lenguaje de programación OO. Su foco principal es mantener la cohesión alta, el acoplamiento bajo, facilitar los cambios y prevenir errores inesperados.

Son importantes porque son una guía para determinar si estamos generando un software de calidad o no.

Acoplamiento: nivel de dependencia entre clases. Que tanto afecta a una clase que otras cambien.
Cohesión: nivel de relación que existe entre los métodos y atributos que tiene una clase. Que una clase sea cohesiva significa que hace pocas cosas y que esas pocas cosas que se hace se encuentran altamente relacionadas.
Delegación: es cuando una clase le pasa la responsabilidad a otra de hacer una determinada acción, y para ello, le pasa todos los datos necesarios para que la pueda ejecutar, muchas veces incluso se pasa a ella misma como parámetro. (Lo aplican los patrones State y Strategy; está relacionado al principio TDA).

- ***DRY – Don't Repeat Yourself*** (State – Strategy)
  Promueve la reducción de la duplicación, especialmente en programación.
  Según este principio, toda pieza de información nunca debería ser duplicada, ya que la duplicación incrementa la dificultad en los cambios y evolución posterior.
  Hay que apuntar a que cada requerimiento esté en un único lugar, representado por una única porción de código.
  Mejora la mantenibilidad (los ajustes los tengamos que hacer en un solo lugar). Permite crear un código más legible y fácil de entender. Permite la reusabilidad. Mejora el testeo del software; y nos da una mayor velocidad en el desarrollo.

- ***TDA – Tell, Don't Ask*** (State)
  Se apunta a una distribución de responsabilidades equitativa.
  Debemos decirles a los objetos que hagan cosas, y estos objetos internamente tomarán sus propias decisiones según su estado.
  Un objeto le pide a otro que haga algo que necesita y que le devuelva el resultado.
- ***PTI – Program to Interface*** (Observer – Strategy)
  La herencia es un mecanismo para extender la funcionalidad de una aplicación reutilizando la funcionalidad de las clases padre. Permite definir un nuevo tipo de objetos basándose en otro, y obtener

así nuevas implementaciones casi sin esfuerzo, al heredar la mayoría de lo que se necesita, de clases ya existentes.
Todas las clases que derivan de una clase abstracta compartirán su interfaz. Esto implica que una subclase simplemente añade o redefine operaciones, y no oculta operaciones de la clase padre. Todas las subclases pueden responder entonces a las peticiones en la interfaz de su clase abstracta.
Busca reducir el acoplamiento entre módulos o sistemas (contexto no defina tantos métodos como clases existan, sino que tenga uno solo apuntando a la interfaz); relacionarse con abstracciones y no con concreciones.
Busca la transparencia con el cliente (los clientes no tienen que saber los tipos específicos de los objetos que usan; los clientes desconocen las clases que implementan dichos objetos).

- ***COI – Composite Reuse Principle*** (Strategy)
  En lugar de que los objetos se autoimplementen un comportamiento, pueden delegarlo a otros objetos. Usar composición nos permite reutilizar comportamiento (código) a través de la delegación (mecanismo de reúso a nivel de objetos), manteniendo un acoplamiento menor. Además, el cliente no conoce el interior de la clase contenida, respetando el encapsulamiento.

- ***EWV – Encapsulate What Varies*** (State – Strategy)
  Busca separar del código las partes propensas a cambiar y encapsularlas en otros objetos, para que cada vez que haya que hacer un cambio, el impacto esté localizado en una pequeña y única parte del sistema. Mejora la flexibilidad y mantenibilidad del sistema, y permite manejar la variabilidad

PRINCIPIOS SOLID

- ***SRP – Principio de responsabilidad única*** (Adapter – State – Iterator)
  Cada clase debería tener una única responsabilidad, un único motivo para cambiar.
  Intenta hacer a cada clase responsable de una única parte de la funcionalidad proporcionada por el software.

- ***OCP – Principio de abierto-cerrado*** (Adapter – Observer – State – Strategy – Iterator)
  Las clases (entidades de sw) tienen que estar cerradas para modificaciones y abiertas para extensiones de funcionalidad (agregar nuevas clases).
  La clase se crea y se cierra; si hay modificaciones en los requerimientos, se colocan nuevas clases con referencia a esa clase principal. No se colocan atributos o métodos a la clase ya creada, se incorporan a las nuevas clases.
  Ante la necesidad de un cambio, no debería cambiar el software existente, sino agregar algo nuevo que trabaje con el anterior sin modificar su funcionamiento.
  Otorga la máxima flexibilidad con el mínimo impacto.

- ***LSP – Principio de sustitución de Liskov*** (State – Template Method)
  (ayuda a predecir si una subclase es compatible con el código que funcionaba con objetos de la superclase).
  Evalúa si la estructura de la herencia está bien diseñada. Si el padre se ubica en el lugar del hijo y más o menos funciona, está bien diseñada/implementada. (Los subtipos deberían poder reemplazarse por sus tipos base sin romper el sistema; el hijo se debe comportar como el padre).

- ***ISP – Principio de segregación de interfaces*** (Adapter – Observer – Strategy)
  No forzar a los clientes a depender de métodos que no utilizan (interactuar o relacionarse con interfaces muy complejas, con cosas que no necesita), ni forzar a las clases del cliente (clases concretas) z implementar comportamientos que no necesitan.
  Se deben desintegrar las interfaces “gruesas” hasta crear otras más detalladas y específicas. Los clientes deben implementar únicamente aquellos métodos que necesitan de verdad.

❖ ***DIP – Principio de inversión de dependencia*** (Template Method)
No debemos hacer depender a clases de un nivel superior de clases de un nivel inferior.
Hay que desacoplar poniendo una interfaz en el medio (las clases de alto nivel van a depender de esa interfaz, en vez de depender de clases concretas de bajo nivel), entonces los módulos de bajo nivel proveen los servicios que requiere el módulo de alto nivel.

PATRONES DE DISEÑO

Son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software; es un concepto general para resolver un problema particular.

Un patrón es una solución concreta a un problema bien definido.

Son importantes porque ayudan a centralizar el conocimiento y porque ayudan a resolver un problema común de una manera mucho más eficiente, ya que no tenemos que pensar la solución a un determinado problema todas las veces que surja. Nos enseñan a resolver todo tipos de problemas utilizando principios de diseño.

¿A qué ayudan los patrones?

1) Encontrar objetos:
Ayudan a identificar abstracciones (clases como la estrategia o estados) no tan obvias, implementadas mediante clases de fabricación pura, y que son fundamentales para lograr un diseño flexible, bajar el acoplamiento y aumentar la cohesión.

2) Determinar la granularidad: (que tanto va a hacer una clase)
Cuánto comportamiento tienen las clases. Las clases de granularidad fina son las que implementan pocos métodos, mientras que las clases de granularidad gruesa son las que implementan muchos métodos.
El nivel de cohesión óptimo para las clases, es el que manejan las clases de granularidad fina, ya que, relacionándolo con el principio de responsabilidad única, buscamos hacer a cada clase responsable de una única parte de la funcionalidad proporcionada por el software, y que esa responsabilidad quede totalmente encapsulada en la clase.
Los patrones nos van a ayudar a decidir cuántos métodos implementamos en cada clase.

3) Especificar interfaces:
Interfaz → tipo especial de clase abstracta en la que todos sus métodos están vacíos.
Permiten ayudar a encontrar interfaces que van a contener el conjunto de peticiones que pueden realizar los clientes, y que ayudan a que estos no tengan que preocuparse por las distintas clases concretas que se van a encargar de esas peticiones.
Ayudan a *programar hacia la interfaz y no hacia la implementación*, lo cual favorece a que las soluciones que apliquemos sean transparentes y fáciles de entender para el cliente, aunque sean más difíciles programarlas.

4) Especificar implementación:
(hablamos de código que va dentro de los métodos. Hay ciertos patrones que ayudan a saber qué métodos debemos usar o qué algoritmos utilizar en los métodos).
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.
Los patrones nos dan pautas para definir las clases participantes; nos ayudarán a entender la diferencia entre la herencia de clases y la herencia de interfaces.
La herencia de clases es un mecanismo para extender la funcionalidad de una aplicación reutilizando la funcionalidad de las clases padres; nos permite definir un nuevo tipo de objetos basándose en otros, heredando lo que necesita. Y la herencia de interfaces, describe cuándo un objeto se puede usar en el lugar del otro.

Teniendo en cuenta el PTI, manipular objetos en términos de la interfaz definida por las clases abstractas tiene dos ventajas: los clientes no tienen que conocer los tipos específicos de los objetos que usan; y los clientes desconocen las clases que implementan dichos objetos.
No se deben declarar las variables como instancias de clases concretas, sino que se ajustarán simplemente a la interfaz definida por una clase abstracta.

5) Favorecer la reutilización:
Pueden favorecer dos tipos de reúso; reúso de caja negra (los detalles internos de los objetos no son visibles) y reúso de caja blanca (las interioridades de las clases padres suelen hacerse visibles a las subclases)
Se trata de favorecer reúso de caja negra (composición) por sobre el reúso de caja blanca (herencia).
Optar por la composición de objetos frente a la herencia de clases ayuda a mantener cada clase encapsulada y centrada en una sola tarea.

6) Diseñar para el cambio:
Se debe diseñar el software de forma que sea flexible, robusto y fácil de evolucionar, ya que los requerimientos cambian. Los patrones nos ayudan a estar preparados para que hayan cambios en el software, y para que los podamos implementar al menor costo posible.

PATRONES DE CREACIÓN

Abstraen el proceso de creación y ocultan los detalles de cómo los objetos son creados, mejorando y flexibilizando tal proceso a través de la distribución de responsabilidades.

- **Singleton**: Garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a ella. (Nos permite acceder a un objeto desde cualquier parte del programa).
  Usar el patrón cuando en el programa necesitas tener una sola instancia disponible para todos los clientes (ej, un único objeto de base de datos compartido por diferentes partes del programa); o cuando necesitas un control estricto sobre las variables globales.
  - SRP: Puede cumplir con este principio si sólo se utiliza para una única responsabilidad, como administrar un recurso compartido, como lo puede ser la conexión con la base de datos.

PATRONES DE ESTRUCTURA

Determinan cómo combinar objetos y clases para definir estructuras complejas. Explican cómo ensamblar objetos y clases en estructuras más grandes.

- **Adapter**: Intermediario entre dos clases que no pueden comunicarse. Cuando existen clases incompatibles, adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
  - SRP: Puede separar la interfaz (o el código de conversión de datos) de la lógica comercial principal del programa.
  - OCP: Se pueden introducir nuevos tipos de adaptadores en el programa, sin romper el código del cliente (el que necesita la información) existente.
  - ISP: La interfaz intermedia tiene los comportamientos que le van a servir al cliente y los que tiene que implementar el adaptador.

PATRONES DE COMPORTAMIENTO

Manejan la comunicación entre objetos del sistema y el flujo de información entre ellos. Se encargan de delegar y distribuir las responsabilidades entre los objetos.

- **Observer**: Define una dependencia de uno a muchos entre objetos, de forma que cuando cambie el estado de un objeto, se notifica y se actualizan todos los objetos que dependen de él.

- OCP: Se pueden agregar nuevas clases suscriptoras sin tener que cambiar el código del publicador (y viceversa si hubiera una interfaz publicador).
- ICP: Tengo dos interfaces; pongo el comportamiento que tiene que ver con los sujetos en la ISujeto, para que los sujetos concretos lo implementen, y con el observador igual.
- PTI: Reducimos el acoplamiento entre sujetos y observadores porque se comunican a través de la interfaz.

- **State**: Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.
  - SRP: Organiza el código relacionado a estados particulares en clases separadas.
  - OCP: Se pueden agregar nuevos estados sin cambiar las clases estado existentes o el contexto.
  - LSP: Los Concretos se pueden intercambiar con el Abstracto, ya que todos son capaces de responder a sus métodos.
  - DRY: El código común se mantiene en la implementación trivial del padre, evitando redefinirlo en todas las hijas.
  - EWV: Los concretos encapsulan la lógica específica de cada uno, aislando los cambios del resto, sin afectarlos.
  - TDA: El contexto delega el trabajo al Estado, que resolverá la lógica del pedido.

- **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 usa.
  - OCP: Se pueden agregar nuevas estrategias sin tener que cambiar el contexto.
  - ISP: Se relaciona una clase concreta con una interfaz que la implementa. La interfaz le da a las clases concretas los métodos que necesita para implementar.
  - DRY: La parte común del algoritmo se mantiene en el contexto, evitando duplicaciones.
  - EWV: Las partes del algoritmo que varían se encapsulan en las Estrategias.
  - PTI: Se establece una interfaz única y todas las estrategias se implementan hacia ella para utilizarse en el contexto.
  - COI: Se usa composición para que el contexto mantenga referencia a las estrategias.

- **Template Method**: Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.
  - DIP: En pos de definir una estructura de algoritmo que se respete y no se cambie, puede ocurrir que el padre tenga una invocación concreta de una clase hija.
  - LSP: Analiza si está bien armada la herencia.

- **Iterator**: Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna. (Extraer el comportamiento de recorrido de una colección y colocarlo en un objeto independiente llamado Iterador).
  - SRP: Puede limpiar el código del cliente y las colecciones extrayendo algoritmos de recorrido voluminosos en clases separadas.
  - OCP: Puede implementar nuevos tipos de colecciones e iteradores, y pasarlos al código existente sin romper nada.
  - EWV: Un objeto iterador encapsula todos los detalles del recorrido. (Varios iteradores pueden recorrer la misma colección al mismo tiempo, independientemente unos de otros).
  - PTI: Todos los iteradores deben implementar la misma interfaz. (Código cliente sea compatible con cualquier tipo de colección o cualquier algoritmo de recorrido).
  - ISP: Al ofrecer una interfaz específica y limitada para la iteración.
  - DIP: Al depender de abstracciones en lugar de implementaciones concretas.
  - LSP: Al permitir que diferentes iteradores se comporten de forma intercambiable.

¿Qué se diseña?

- *Arquitectura*: Es el conjunto de decisiones significativas respecto a resolver requerimientos no funcionales y funcionales. Es diseño general, hay definiciones de cómo resolver requerimientos no funcionales frente a un contexto de infraestructura específico.
  Toma los requerimientos no funcionales significativos para la arquitectura y los aplica al modelo de análisis.
  Se representa a través de vistas, es un modelo genérico que no entra en detalles. La funcionalidad condiciona a la arquitectura; avanzan juntas, ya que se condicionan entre sí.
- *Procesos*: Transforma elementos estructurales de la arquitectura del programa en una descripción procedimental de los componentes del sw. Se utilizan patrones de diseño para la creación de procesos. A partir de las realizaciones de casos de uso de análisis se obtienen las realizaciones de casos de uso de diseño luego de considerar y aplicarles los RNF.
  Cómo vas a hacer para distribuir los componentes de sw en el hardware para cumplir con los requerimientos funcionales y no funcionales del sistema.
- *Datos*: Implica analizar nuestras clases y definir cuales de esas clases necesitan persistencia, en qué almacenamiento persistente las vamos a guardar, diseñar la BD y la estructura de los datos, y diseñar la persistencia (como hago para conectar las clases con la BD). Transforma los requerimientos en las estructuras de datos necesarias para hacer persistente el sw.
- *IHM*: Es una disciplina que se encarga de evaluar todos los aspectos que tienen que ver en cómo interactúan las personas con el sw. Esta disciplina combina las interfaces de usuario y la experiencia de usuario. Una tiene que ver con cómo ve el usuario el sistema, y la otra tiene que ver con cómo efectivamente funciona el sistema y cómo lo percibe el usuario.
- *Entrada/Salida*: Describe cómo se ingresa información al sw y cómo se presentarán las salidas del mismo.
  En lotes →se acumulan las transacciones para procesarlas después.
  En línea →en el momento en que ocurre la transacción, ésta es procesada.
  En tiempo real →son sistemas en línea, pero pueden modificar el ambiente donde está inmerso, tienen un tiempo estricto de respuesta.
- *Procedimientos manuales*: Describe cómo se integra el sw al sistema de negocio. Incluye los 'plan B' en caso de que el mismo falle. Nos permite insertar el sw en el negocio.

DISEÑO DE PERSISTENCIA

Persistencia → capacidad que tiene un objeto de permanecer en el tiempo y en el espacio.

DISEÑO DE PERSISTENCIA

Implica analizar nuestras clases, definir cuáles de esas clases necesitan persistencia, en qué almacenamiento persistente las vamos a guardar, diseñar la base de datos y la estructura de los datos, y diseñar la persistencia (cómo conectar las clases con la base de datos).

La persistencia la hemos hecho presente en la arquitectura como un componente "persistencia" que hacía de vínculo entre la lógica de negocio, que debía ser mantenido por un tiempo, y la base de datos relacional.

El origen de la problemática de necesitar del diseño de persistencia son los paradigmas, ya que el programa se diseña según el POO, y la base de datos trabaja con el paradigma relacional o estructurado.

- Hay que determinar qué objetos necesitan persistencia. Los objetos de entidad son los candidatos importantes para el almacenamiento persistente.
- (Es cierto que existen bases de datos de objetos, y utilizando este tipo de bases de datos no se necesita de ningún servicio de persistencia, pero no son las bases de datos más utilizadas; generalmente la mayoría de bases de datos que se utilizan en los programas son relacionales).
- Bases de datos relacionales: En este caso surgen problemas de incompatibilidad entre la representación de los datos orientada a registros y la orientada a objetos, por lo que se requiere un servicio especial para establecer la correspondencia.

- También podemos querer almacenar los datos en formato xml, en bases de datos jerárquicas, etc. De igual manera vamos a necesitar un servicio especial que haga que funcionen con objetos.

Problema de impedancia: Ocurre cuando queremos guardar objetos a BD relacionales. El primer problema es que las BDR no soportan comportamiento, sólo guardan datos; y el segundo problema es que no puedo guardar cualquier tipo de datos, porque sólo permiten primitivos
Se refiere al problema de la correspondencia entre los paradigmas. En el paradigma OO la información se almacena en los objetos, y las bases de datos relacionales pertenecen al paradigma estructurado. Por lo tanto, necesitamos transformar nuestra estructura de información de objeto a una estructura orientada a tablas. Esto crea un fuerte acoplamiento entre la aplicación y el DBMS.

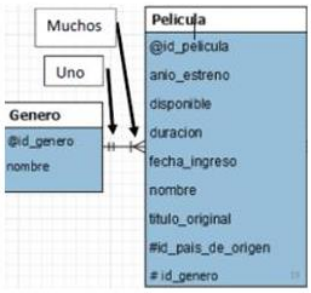
- Las bases de datos relacionales no soportan comportamiento, los métodos no son soportados, todos los métodos que hacen la integración de la clase y que es lo más importante que tiene la clase no se puede guardar en una base de datos relacional.
- Las bases de datos relacionales sólo almacenan tipos de datos primitivos y no objetos complejos, estos tipos de datos los define el proveedor del motor de base de datos, y no se pueden definir tipos propios de datos y las clases no son ni más ni menos que un tipo de datos propio. Hay que desarmar la estructura de clases y mapearlas a cada una de las columnas de las tablas.

La vinculación entre el modelo de clases y la base de datos relacional es el MAPEO o diseño de persistencia.

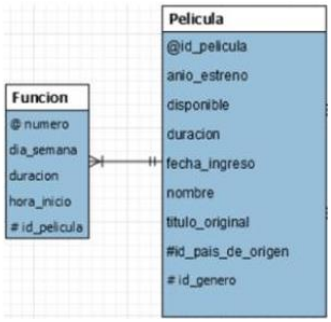
Las clases persistentes se les asigna una tabla y se les descarta el comportamiento. A cada atributo se lo coloca en una columna de la tabla. La tabla se llama igual que la clase para mantener la trazabilidad o el vínculo. Cada atributo primitivo se transformará en una columna en la tabla. Si el atributo es complejo agregamos una tabla adicional o distribuimos el atributo en varias columnas de la tabla de la clase. 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.

- Opciones para el mapeo de asociación

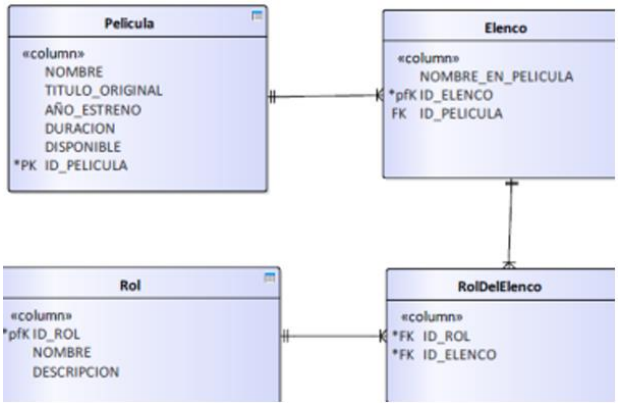
1. 1 a muchos



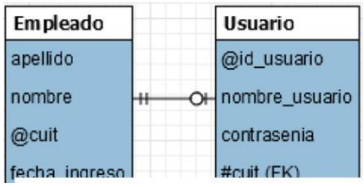
2. Muchos a uno



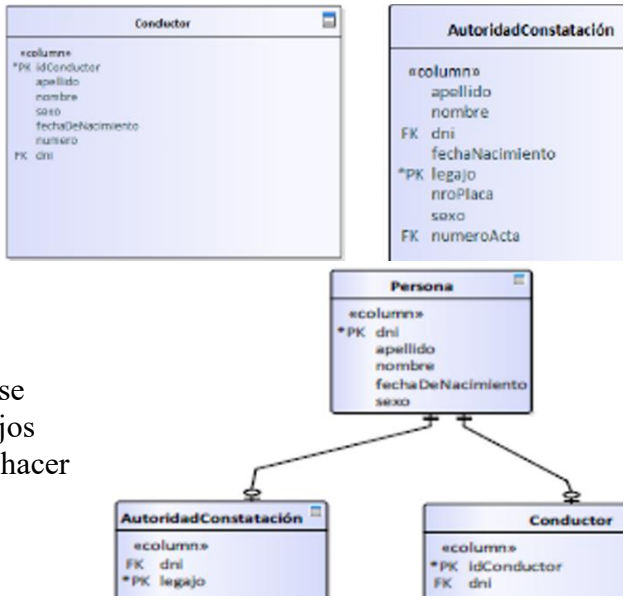
3. *Muchos a muchos*: se crea una tabla intermedia asociativa. Es necesaria por el problema de la clave foránea.



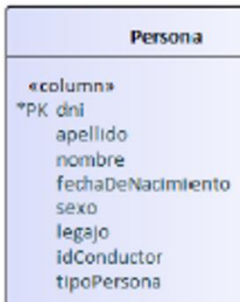
4. *Uno a uno*: las relaciones uno a uno de ambos lados no debería existir porque en ese caso debería fusionar todo en una sola tabla. En este caso si se puede porque del lado del usuario tenemos opcionalidad 0..1.



- Opciones para el mapeo de la herencia
  1) *Eliminar al padre*: significa eliminar la herencia. Quedan 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 y no hay que hacer joins. El tamaño de la tabla aumenta ya que se duplican columnas (los atributos comunes van a repetirse). Puede traer problemas de consistencia y redundancia.



  2) *Simularla*: es una jerarquía supertipo-subtipo. La clase abstracta está en su propia tabla y las tablas de los hijos hacen referencia a ella. Reduce la redundancia, pero hacer JOINs que pueden reducir la performance.



3) *Eliminar los hijos*: se ponen todos los atributos de todas las clases hijas en una sola tabla. No se recomienda porque no hay trazabilidad, hay baja cohesión y se desperdicia almacenamiento ya que muchos atributos quedarán en blanco.

Para elegir entre las distintas opciones tengo que analizar el volumen de las tablas, la cantidad de registros y cómo va a ir creciendo el modelo.

¿a qué clase de nuestro software le vamos a dar la responsabilidad de que haga el trabajo de hacer las clases persistentes? → por esto se utiliza un framework de persistencia.

<u>Framework de persistencia</u>

Es un conjunto de tipos de propósito general, reutilizable y extensible que proporciona funcionalidad para dar soporte a los objetos persistentes.

Un servicio de persistencia o subsistema realmente proporciona el servicio, y se creará con un framework de persistencia.

Un framework de persistencia contiene clases concretas y abstractas que definen interfaces a las que ajustarse, interacciones de objetos en las que participar, y otras variantes. Ofrece un alto grado de reutilización. Debería proporcionar funciones para almacenar y recuperar objetos en un mecanismo de almacenamiento persistente, y confirmar y deshacer las transacciones.

Generalmente, un servicio de persistencia tiene que traducir los objetos a registros y guardarlos en una base de datos, y traducir registros a objetos cuando los recuperamos de la base de datos.

- Materialización: Una o varias filas de tablas de la base de datos son convertidas 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 de tablas.

Los modelos de persistencia que tenemos son:

Superobjeto persistente

Se propone crear una clase de fabricación pura (SuperObjetoPersistente), y en esa clase definir el método de materialización y desmaterialización, junto con cualquier método que necesitemos para resolver la persistencia.

Entonces, todas las clases que necesiran persistencia deberían heredar de la clase SuperObjetoPersistente.

Se llama correspondencia directa porque no hay nada en el medio entre la clase de fabricación pura y las clases que necesitan persistencia. El SuperObjetoPersistente define los comportamientos, además de tener la responsabilidad de saber cómo hacerse persistentes en una base de datos.

Ventajas:

- ✓ No crece la estructura de clases.
- ✓ Fácil de implementar.

Desventajas:

- ✓ Si la clase trae herencia del dominio y también tiene que heredar la persistencia, hay muchos lenguajes de programación que no soportan la herencia múltiple, sólo herencia simple. Aunque se podría solucionar con una realización de interfaz para el SuperObjetoPersistente.
- ✓ No se cumple e principio de diseño open-closed, ni tampoco el principio de responsabilidad única. Una clase de dominio no debería saber cómo hacerse persistente porque no modela ese concepto.
- ✓ Se rompe la cohesión.
- ✓ Desde el punto de vista arquitectónico, se presenta el problema de acoplamiento entre la capa de negocio y la capa de base de datos.

Esquema de persistencia

Se basa en un esquema de correspondencia indirecta. Es lo que se usa desde la arquitectura y es lo que se recomienda. Para que funcione, se agrega en el esquema de persistencia una clase que funcione como intermediaria para clase de negocio. Esas clases tienen el método de materializar y desmaterializar implementado. Además, contienen la PK y el puntero, y se puede mantener la relación entre ambos paradigmas. Le quitamos la responsabilidad a las clases de negocio de saber cómo materializarse y desmaterializarse.

Es una decisión arquitectónica. Agrego una capa (intermedia) de persistencia entre la capa Lógica de Negocios y la BD, compuesto por clases de fabricación pura, que tienen el rol de establecer vínculos entre la capa de Lógica de Negocio y la capa de Datos, sin que los objetos se vean afectados por el problema de persistencia. El esquema de persistencia resuelve el problema de materializar, desmaterializar y los métodos necesarios para la persistencia.

El esquema de persistencia tiene un conjunto de clases o interfaces que definen los métodos, y para integrar este esquema a mi modelo de dominio, tengo que crear una clase de fabricación pura por cada clase del dominio que necesite persistencia.

Ventajas:

- ✓ Se respeta el principio de responsabilidad única y open-closed porque las clases de dominio no se tocan.
- ✓ Mantiene alta la cohesión.
- ✓ Soporta extensibilidad.
- ✓ Mejora la reusabilidad.
- ✓ Mantiene bajo el acoplamiento, ya que aplica el principio de inversión de dependencias. El control lo toman las clases del esquema de persistencia, que van y buscan a los objetos, y los materializan y desmaterializan.

Desventajas:

- ✓ Hay un aumento considerable de clases.

Patrones de persistencia

- *Patrón identificador de objetos*: Propone asignar un identificador de objeto a cada registro y objeto.
- *Patrón de representación de objetos como tablas:* Propone la definición de una tabla en una BDR por cada objeto persistente.
- *Patrón fachada:* Permite el acceso al servicio de persistencia mediante fachada, proporcionando una interfaz uniforme a un subsistema. La fachada no hace el trabajo, sólo lo delega en objetos del subsistema.
- *Patrón conversor o broker:* Propone crear una clase responsable de materializar y desmaterializar los objetos. Agrega un intermediario por cada clase de dominio.
- *Materialización con el método plantilla:* Con el patrón template method encapsulamos en una sola clase el comportamiento común de la materialización. El punto de variación es la manera de crear el objeto a partir del almacenamiento.
- *Patrón de gestión de caché:* Es conveniente mantener los objetos materializados almacenados en una caché local para mejorar el rendimiento y dar soporte a las operaciones de gestión de transacciones, con un OID como clave. El conversor primero busca en la caché para evitar materializaciones innecesarias.
- *Estados transaccionales y el patrón state:* Los objetos persistentes pueden insertarse, eliminarse o modificarse. Operar sobre un objeto persistente no provoca una actualización inmediata en la base de datos, más bien se debe ejecutar una operación commit explícita.
- *Diseño de transacción con patrón command:* Con el patrón se busca crear un objeto por cada operación a realizar en la transacción, haciendo una cola de operaciones permitiendo hacer un rollback, si es necesario.
- *Materialización perezosa mediante proxy virtual:* Se basa en la suposición de que los proxies conocen el OID de sus sujetos reales, y al requerir la materialización, se lo usa para identificar y recuperar el sujeto real.

UNIDAD 3

ARQUITECTURA

Conjunto de decisiones significativas que tomamos respecto de cómo resolver los requerimientos no funcionales (o de calidad), teniendo en cuenta el contexto (reglas de negocio) donde va a funcionar el sistema.

Los sistemas se descomponen en subsistemas. El proceso de diseño para identificar estos subsistemas y establecer un marco de trabajo para control y comunicación de los subsistemas se llama “diseño arquitectónico”.

DISEÑO ARQUITECTÓNICO

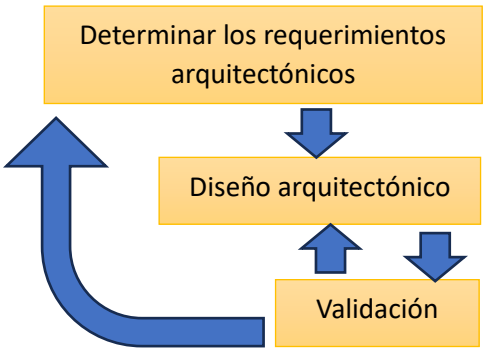
Diseño estratégico que se encarga de la asignación de modelos de requerimiento esenciales a una tecnología específica. Proceso que toma el modelo de análisis y los requerimientos no funcionales que no tuvimos en cuenta antes, y los aplica sobre una tecnología específica; planifica a un nivel alto de abstracción, lo que va a ser el diseño y la implementación del software.

Se interesa por entender cómo debe organizarse un sistema y cómo tiene que diseñarse la estructura global de ese sistema.

Es el “plan de diseño” → donde se toman las decisiones de cómo vamos a resolver esos requerimientos.

La arquitectura captura la estructura del sistema en términos de componentes y cómo éstos interactúan.

PROCESO DE DISEÑO DE LA ARQUITECTURA DE SOFTWARE



La arquitectura se diseña porque en esta es donde se ejecuta el sw, los RF se ejecutan en una determinada arquitectura.
La arquitectura es la base en donde se ejecutan las distintas funcionalidades del sistema.; debo diseñar cómo es que cada una de las funcionalidades se va a ejecutar. Los CU se van a ejecutar en una determinada arquitectura, entonces tengo que pensar en qué arq es más conveniente que se ejecuten, sobretodo pensando en los RNF.

El proceso de diseño arquitectónico es iterativo e incremental, porque no es que diseñas una vez la arquitectura y después ese diseño no se modifica más, sino que lo vas actualizando a medida que vas incorporando conocimientos sobre el dominio y que vas validando lo que hiciste.

Determinar los requerimientos arquitectónicos

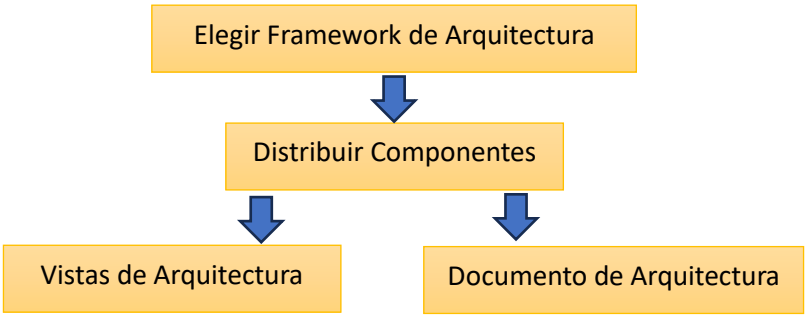
Se trata de diseñar un modelo de requerimientos que nos conduzca al diseño arquitectónico y su priorización. Los requerimientos arquitectónicos son esencialmente la calidad y los requerimientos no funcionales de la aplicación.

Primer paso: *Identificar los requerimientos arquitectónicos*
Se toma como entrada los requerimientos funcionales y los requerimientos de los involucrados. Se determinan los requerimientos arquitectónicos, y se produce como salida un documento que contiene los requerimientos arquitectónicos de la aplicación.

Segundo paso: *Priorizar requerimientos*
(requerimientos relacionados entre sí deben tener la misma prioridad)
Prioridad Alta: La aplicación debe soportar este requerimiento, conducen el diseño de arquitectura; tienen que estar disponibles desde la primera iteración ya que postergarlos implica retrabajo.
Prioridad Media: Necesitará ser soportado en alguna etapa, pero no necesariamente en la primera.
Prioridad Baja: Estos son parte de la lista de deseo. Las soluciones que los incluyen son deseables, pero no son conductores del diseño; pueden ser postergados para las últimas iteraciones.

Diseño arquitectónico

Incluye la definición de una estructura de componentes junto con las responsabilidades que tendrán cada uno de ellos.



1) Elegir framework de arquitectura
(elección de los patrones que voy a aplicar en el producto de software)
Ver de todas las soluciones arquitectónicas del mercado, cuál nos serviría a nosotros para implementar, dado los requerimientos no funcionales y las características del producto. Se elige un framework que puede constar de uno o más patrones arquitectónicos para definir y empezar a diseñar la arquitectura.

2) Distribuir componentes
   Se deben definir los componentes principales que comprenderán el diseño. Se conducirá a distribuir los componentes teniendo en cuenta qué componentes se van a utilizar, las dependencias entre ellos, la información que van a manejar, etc.

   Produce como salidas las vistas arquitectónicas, que se utilizan para modelar/reflejar la arquitectura; y un documento de arquitectura que sirve para documentar el producto de software (es un lugar donde voy poniendo las decisiones que tomo para construir la arquitectura de software y el por qué, para dejar un registro de esas decisiones)

Validación

Es probar la arquitectura, comúnmente recorriendo el diseño, contra los requerimientos existentes y los que se pueden predecir en un futuro. Sirve para mostrar que la propuesta arquitectónica realmente resuelve los requerimientos. (arquitectura cumpla su propósito)
Hay dos técnicas principales para la validación:

- Uso de escenarios: Consiste en un testeo manual de la arquitectura usando casos de prueba. Están relacionados con aspectos arquitectónicos tales como atributos de calidad, y ellos ayudan a destacar consecuencias de las decisiones arquitectónicas que están encapsuladas en el diseño. (cuenta con una serie de casos de prueba que validarán el sistema en cada uno de los contextos de los casos de pruebas).
- Prototipos: Es una primera aproximación del sistema funcionando, ayuda a brindarle al usuario una primera impresión del sistema. Un prototipo sirve para probar las distintas funcionalidades que tiene un sistema. Tiene la capacidad de evaluar los requerimientos en forma detallada.

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.

Documentación de la arquitectura

La descripción de la documentación se hace a través de las vistas (la arquitectura se modela a través de vistas y se construyen en base a la identificación y priorización de los RNF significativos), las cuales se construyen con los diagramas UML. Además, contiene las distintas decisiones de diseño que se tomaron y que el resto del equipo deben respetar, además de las definiciones generales sobre cómo se va a construir el producto.

Es importante documentar porque sirve para la comunicación entre los distintos participantes; sirve para hacer análisis y determinar si se cumplen los requerimientos críticos; sirve para tener un registro de las decisiones que se tomaron y por qué éstas fueron tomadas; a través de las vistas puedo ver los distintos puntos de vista y perspectivas del sistema, y puedo centrarme sólo en lo que me interesa (despliegue, implementación…).

Vistas de la arquitectura

Las vistas están pensadas para un involucrado en particular y sus intereses.

Modelan un punto de vista, que es una descripción teórica de lo que se va a mostrar en esa vista. Cuando modelo una vista, es más lo que no muestro que lo que muestro, ya que debo elegir qué es lo que tengo que mostrar para un interés particular, y es imposible mostrar todos los detalles de un sistema en una vista. Por eso es que existe distintas vistas que muestra distintos puntos de vista (perspectivas).

Son utilizadas para manejar la complejidad y para poder comunicar de una manera mucho más sencilla.

Vistas 4+1 (en la U1 también están explicadas)

Estas 4 vistas están unidas por la Vista de Casos de Uso (Vista de la funcionalidad), la cual contiene los casos de uso significativos para la arquitectura (capturan los requisitos para la arquitectura). A partir de esta vista se van a construir las demás, respetando la característica del PUD de “conducido por casos de uso”.

- Vista de diseño (lógica): describe los elementos significativos de la arquitectura y las relaciones entre ellos. Describe cómo será provista la funcionalidad del sistema, qué clases, objetos o componentes voy a utilizar para darle soporte a la funcionalidad. La vista lógica captura la estructura de la aplicación.
- Vista de proceso: Describe la concurrencia y los elementos de comunicación de una arquitectura.
- Vista de despliegue (física): Describe cómo los principales procesos y componentes se asignan al hardware de las aplicaciones. Muestra cómo el software será alojado en los diferentes componentes de hardware.
- Vista de implementación (componentes): Captura la organización interna de los componentes de código, normalmente cuando se mantienen en un entorno de desarrollo o herramienta de gestión de la configuración. Describe también las dependencias de los módulos de implementación (código fuente, bibliotecas, componentes de terceros).

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. Diagrama de módulo, de casos de uso. Patrón Layered.
- Vista de ejecución: Contiene vistas de los elementos que se pueden ver en tiempo de ejecución. Incluye escenarios de funcionalidad, vistas de responsabilidad y ensambles de componentes. Instancias de componentes, conectores y puertos. Diagrama de contexto del sistema. Patrón Messaging, publish & suscribe, broker y process coordinator.
- Vista de distribución: Contiene vistas de los elementos relacionados con la distribución del sw en el hw. Diagrama de distribución. Patrón n-tier, map reduce, mirrored, granja, rack.
- Vista de spanning: Contiene cuestiones de confiabilidad, seguridad, performance, etc. (escenarios de los atributos de calidad).

Se pueden agregar otras vistas como la vista de datos o de seguridad. La vista de datos se puede incluir, ya que, al estar trabajando con una BD relacional, es útil mostrar qué decisiones se toman respecto a los datos para pasarlos del paradigma OO al paradigma relacional o modelo de datos relacional. La vista de seguridad puede incluirse si ésta es un requerimiento muy crítico para la aplicación, entonces ésta vista muestra cómo se resuelven los aspectos de seguridad.

Hay que recordar que las vistas son un diseño de alto nivel que nos muestran de manera general como queremos que nuestro producto resuelva los RF y los RNF.

PATRONES ARQUITECTÓNICOS

Es una descripción abstracta de buena práctica, que se ensayó y se puso a prueba en diferentes sistemas y entornos. Debe describir cuando es y no es adecuado usarlo, así como sus fortalezas y debilidades.

Un framework arquitectónico es el conjunto de patrones arquitectónicos que se utilizan para estructurar una aplicación.

No se contrapone a la idea de que hay un estilo arquitectónico predominante.
Patrones platónicos →son los que se ven en los libros y rara vez se implementan de esa forma en el código.
Patrones embebidos →son los que se ven en los sistemas reales y suelen violar las restricciones estrictas de los platónicos, y generalmente esto se hace para poder cumplir con ciertos requerimientos no funcionales que son críticos.

a. Layered (en capas):

Consiste en una pila de capas en donde cada una actúa como una máquina virtual de la capa de arriba. Es un estilo estratificado simple, las capas sólo pueden usar la capa que está directamente debajo de ellas. La restricción significa que las capas subsiguientes de más abajo se encuentran ocultas.

Es 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 que le ofrece la capa inmediata inferior (las capas sólo pueden ver las que se encuentran inmediatamente debajo de ellas).

La calidad resultante es que trata de manera directa 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. Las capas superiores dan más oportunidades de sustitución frente a un posible problema de performance.

Existen variantes en donde las capas pueden comunicarse con capas de más abajo (que no son adyacentes).

Otra variante es el uso de capas compartidas, en donde cada capa puede usar capas verticales.

Este patrón puede variar de forma considerable 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. Aún así, es beneficiosa dado que las capas agrupan módulos de funcionalidad coherente.

El propósito de este patrón es lograr el mejor nivel de cohesión y acoplamiento posible, decidiendo en qué capa voy a ubicar cada componente de software.

Se usa al construirse nuevas funcionalidades encima de los sistemas existentes, cuando el desarrollo se dispersa a través de varios equipos de trabajo, cuando existe un requerimiento de capa multinivel.

Ventaja: Permite sustituir una capa completa por otra sin modificar ninguna de las capas adyacentes, siempre y cuando la interfaz no cambie. En cada capa pueden incluirse facilidades redundantes para aumentar la confiabilidad.

Desventaja: En la práctica suele ser difícil ofrecer una separación limpia entre capas, donde posiblemente una capa de nivel superior tenga que interactual con una capa de nivel inferior no adyacente. El rendimiento suele ser un problema debido a los múltiples niveles de una solicitud de servicios mientras se procesa en cada capa.

Una capa es el nombre que se le da desde la arquitectura a un subsistema. Dentro de una capa hay componentes.

Forma parte de la vista de módulo. Muestro los componentes y donde los voy a ubicar, pero no se muestra la comunicación o vínculo. Se ve el sw desde un punto de vista estructural, no se ve la ejecución.

b. Publish & Suscribe

Funciona cuando hay un evento que hace que cambie de estado un determinado publicante.

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

Los publicantes ignoran el motivo por el cual el evento es publicado, los suscriptores ignoran el por qué o quién 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 desuscribirse dinámicamente de un tópico.

Es parte de la vista de ejecución.

Propiedades claves del patrón→mensajería mucho a muchos, calidad de servicio configurable, bajo acoplamiento.

Las arquitecturas basadas en este patrón son muy flexibles y adecuadas para aplicaciones que requieren mensajes asíncronos uno a muchos, muchos a uno, o muchos a muchos entre componentes. Puede configurarse como síncrono. Al ser asíncrono no se pierde información si hay pérdida de conexión, la información queda guarda en el tópico y luego se rehace el intento de entregar los mensajes tantas veces como esté configurado, ya que este patrón funciona con middleware.

Si se quiere garantizar disponibilidad, en este caso, se debe replicar el tópico para que haya redundancia.

En cuanto al tiempo de respuesta, si hay comunicación punto a punto, el proceso de notificar se vuelve más lento que si la comunicación fuese multicast. Por esto es que este patrón está pensado para comunicaciones multicast que es mucho más eficiente en términos de desempeño y de demora en la cantidad de mensajes que pueden entregarse a los suscriptores por unidad de tiempo.

El principal beneficio es desacoplar los componentes que producen los eventos de quienes los consumen, lo que hace que la arquitectura sea más mantenible y evolucionable.

Los componentes pueden asumir el rol de publicantes, de suscriptores o eventualmente ser al mismo tiempo publicantes para un tópico y suscriptores para otro.

Los buses de eventos varían en las propiedades que soportan→durables, entrega en orden, entrega en lotes.

Se usa en arquitecturas muy flexibles y adecuadas para aplicaciones que requieren mensajes asíncronos uno a uno, uno a muchos o muchos a muchos componentes.

Ventaja: Bajo acoplamiento, ya que no existe un vínculo directo entre publicantes y suscriptores, el vínculo se da a través del tópico. Desacopla los componentes que producen los eventos de quienes los consumen, haciendo que la arquitectura sea mantenible y evolucionable.

Desventaja: El bus de eventos agrega una capa de indirección entre productores y consumidores, pudiendo afectar la performance del sistema.

c. Messaging

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

La cola es el componente que acumula peticiones que van a ser entregadas en distintos componentes de sw que necesitan recibir esa información.

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

Pertenece a la vista de ejecución.

Propiedades clave→comunicaciones asíncronas, calidad de servicio configurable, bajo acoplamiento.

Provee una solución resistente para aplicaciones en las cuales la conexión es transitoria, debido a la poca confiabilidad de la red y de los servidores. En este caso, los mensajes se mantienen en la cola hasta que el servidor se conecta y los elimina. Es apropiada cuando el cliente no necesita respuesta inmediata después de enviar el mensaje.

Se puede implementar para pedido respuesta síncrono utilizando un par de colas pedido respuesta.

Se deben acumular peticiones, las cuales serán procesadas con algún criterio de ordenamiento. Se usa cuando no se quiere perder información, ya que está pensado para comunicaciones asíncronas, si se produce una interrupción, la información queda acumulada como petición en la cola, no se pierde y cuando se reestablece la conexión se efectúa la entrega de peticiones.

Si quiero garantizar la alta disponibilidad se debe eliminar la cola.

Ventaja: Promueve el bajo acoplamiento, permitiendo alta modificabilidad, dado que emisores y receptores no se conectan.

Desventaja: Los cambios en los formatos de los mensajes de los receptores pueden causar cambios en las implementaciones de los emisores, pero los formatos de los mensajes auto-descriptivos reducen esta dependencia.

d. Broker

El broker actúa como concentrador de mensajes, y los remitentes y receptores se conectan como radios. Las conexiones al broker son por medio de puertos asociados a un formato específico de mensaje.

El agente incorpora la lógica de procesamiento para entregar un mensaje recibido en un puerto de entrada a un puerto de salida.

La lógica del broker es que transforma el tipo de mensaje de origen recibido en un puerto de entrada, en el tipo de mensaje de destino en el puerto de salida.

Los agentes son adecuados para aplicaciones en las cuales los componentes intercambian mensajes que requieren grandes transformaciones entre los formatos de origen y destino.

El agente desacopla el remitente del receptor, permitiéndolos producir y consumir su formato nativo, y centraliza la definición de transformación lógica en el agente para facilitar la comprensión y modificación.

Se utiliza cuando hay un problema de compatibilidad, y la mayoría de las veces, se encuentra involucrado alguien ajeno a nuestro sistema que no podemos controlar.

Pertenece a la vista de ejecución.

Se debe replicar el broker si se quiere garantizar la alta disponibilidad.

Funciona con comunicaciones asíncronas, y el objetivo es no perder información. Se puede configurar la calidad de servicio para comunicaciones síncronas, aunque no se recomienda.

e. Process Coordinator

El coordinador de procesos encapsula los pasos necesarios para completar un proceso de negocio. El coordinador de proceso es un solo punto de definición para el proceso de negocio, haciéndolo más fácil de modificar y entender. Recibe una solicitud de inicio de proceso, llama a los servidores en el orden indicado por el proceso y emite los resultados.

Soporta bajo acoplamiento, ya que los servidores 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 como parte del proceso de negocio.

La comunicación entre el coordinador y los servidores pueden ser síncronas o asíncronas.

Es utilizado comúnmente para implementar procesos de negocio complejos que deben hacer peticiones a componentes de servidores diferentes. Encapsulando la lógica de proceso en un lugar es mucho más fácil cambiar, administrar y monitorear la performance del proceso.

Tiene una arquitectura centralizada, ya que el propósito de este patrón es ubicar en un componente de software, llamado coordinador de proceso, la responsabilidad de manejar la lógica de un proceso de negocio complejo.

Está pensado para trabajar con comunicaciones asíncronas, aunque se puede configurar para síncronas.

Arquitectura de la vista de ejecución.

Ventaja: Encapsula la lógica de proceso en un lugar, es más fácil de cambiar, administrar y monitorear la performance del proceso. Bajo acoplamiento ya que los servidores no se conocen entre sí. Se configura el coordinador y es el que da la orden de los pasos a seguir. El resultado es único y lo genera el coordinador de proceso.

Desventaja: El coordinador de proceso se puede convertir en un cuello de botella, ya que es un acumulador que debe atender demasiadas peticiones y puede degradar la performance.

f. Client-Server

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 conocen a los servidores y deben saber cómo usarlos, pero los servidores no necesitan conocer a los clientes.

Clientes y servidores son procesos lógicos.

Los clientes inician la comunicación, hacen peticiones y esperan la respuesta.

Los servidores no conocen la identidad de los clientes hasta que se han conectado. Los servidores atienden las peticiones.

El rol de los clientes y servidores no es intercambiable.

Es una arquitectura sincrónica.

Como los servidores se pueden replicar, se usa cuando la carga de un sistema es variable.

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

Es parte de la vista de distribución.

El uso más común es para explotar la capacidad gráfica de la pc, y al mismo tiempo proteger los datos en un host central.

Ejemplos: colas de impresión. Las aplicaciones como Word, Excel son clientes y la cola de impresión de Windows es el servidor.

Algunas variantes son → los conectores pueden ser síncronos o asíncronos; puede haber límite en el número de clientes o servidores; las conexiones pueden ser con o sin estado; la topología del sistema puede ser estática o dinámica; una variante permite al servidor, luego de que el cliente establezca contacto por

primera vez, enviarle actualizaciones posteriores; otra variante es el estilo n-tier que utiliza dos o más instancias del estilo cliente servidor para formar una serie de capas.

- Los pedidos fluyen en una única dirección.
- Las capas tienen responsabilidades funcionales exclusivas.
- Este estilo es un híbrido entre la vista de ejecución y la de distribución, dado que las capas suelen asociarse a componentes de hw distintos.
- Esta variante surge al introducir la arquitectura en las capas, ya que la capa de presentación funciona como cliente, pero las capas intermedias suponen un problema porque deben atender peticiones de las capas superiores y deben hacer peticiones a las capas de abajo. Esto en la arquitectura cliente servidor pura es imposible, por esto es que surge esta variante en donde las capas intermedias pueden funcionar como cliente y como servidor al mismo tiempo, algo que en el cliente servidor puro no es posible.

g. 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 es que toda la red de tuberías está continua e incrementalmente procesando datos.

Se suele utilizar en aplicaciones de procesamiento de datos, donde las entradas se procesan en etapas separadas para generar salidas relacionadas.

Forma parte de la vista de ejecución.

Ventaja: 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.

Desventaja: El formato para transferencia de datos debe acordarse entre los transformadores que se comunican. Cada transformación debe analizar sus entradas y sintetizar sus salidas al formato acordado. Esto aumenta la carga en el sistema y dificulta la reutilización de transformaciones funcionales que usen estructuras de datos compatibles. Son inapropiadas para aplicaciones interactivas.

h. Model View Controller (MVC)

Separa presentación de 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 cómo se presentarán esos datos al usuario. El controlador dirige la interacción del usuario y pasa esas interacciones a vista y modelo, es decir, recibe peticiones de la vista sobre el modelo, interacciona con el modelo y devuelve el resultado a la vista.

Se usa cuando existen múltiples formas de interactuar con los datos. También se utiliza cuando se desconocen los requerimientos futuros de la interacción y presentación.

Forma parte de la vista de módulo.

Ventaja: Permite a los datos cambiar de manera independiente de su representación y viceversa. Soporta diferentes representaciones de los mismos datos.

Desventaja: Puede implicar código adicional y complejidad de código cuando el modelo de datos y las interacciones son simples.

ARQUITECTURA DE SISTEMAS DISTRIBUIDOS

Un sistema distribuido es un sistema en el que el procesamiento de información se distribuye sobre varias 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 sistema coherente.

Tipos de sistemas:

➔ Sistemas personales: No son distribuidos y han sido diseñados para ejecutarse en una computadora personal o estación de trabajo. No comparten recursos de hw.

➔ Sistemas embebidos: Sistemas creados para un hw específico. Está el sw integrado en el hw. Son desarrollados por ingenieros electrónicos o de computación. Se ejecutan en un único procesador o en un grupo integrado de procesadores.

➔ Sistemas distribuidos: El sistema se ejecuta en un grupo de procesadores cooperativos, conectados a través de una red. Son los más utilizados hoy en día y apuntan a la integración.

Ventajas de los sistemas distribuidos

- Compartición de recursos: Permite compartir recursos de hw y de sw. Esto permite aprovechar recursos que no disponemos físicamente y/o están en otro lugar, lo cual aporta a la capacidad de procesamiento.
- Apertura: Son sistemas abiertos, lo que significa que se diseñan sobre protocolos estándares que combinan sw y hw de diferentes vendedores. Esto facilita la integración.
- Concurrencia: Varios procesos pueden estar operando al mismo tiempo sobre diferentes computadoras de la red sin conflictos. Estos procesos pueden comunicarse con otros durante su funcionamiento normal.
- Tolerancia a fallas: La disponibilidad de varias computadoras y el potencial para reproducir información significa que los sistemas distribuidos pueden ser tolerantes a algunos fallos de funcionamiento del hw y del sw. En la mayoría de los sistemas distribuidos, se puede proporcionar un servicio degradado cuando ocurren fallos de funcionamiento, una completa pérdida de servicio sólo ocurre cuando ocurren fallos de funcionamiento en la red.
- 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

- Complejidad: Son más difíciles de comprender y probar. Hay demasiados elementos que pueden fallar. Mover los recursos de una parte del sistema a otra puede afectar de forma radical el rendimiento del sistema.
- Seguridad: Se puede acceder de muchas computadoras, hay mucha vulnerabilidad y muchos elementos para atacar. Esto hace difícil asegurar la integridad de los datos.
- Manejabilidad: Las máquinas pueden ser de distintos tipos y con diferentes sistemas operativos. Los defectos pueden propagarse de una máquina a otra con consecuencias inesperadas, lo que implica que es más difícil de gestionar y administrar, se requiere más esfuerzo.
- Impredecibilidad: Los sistemas distribuidos tienen una respuesta impredecible. La respuesta depende de la carga total en el sistema, de su organización y de la carga de la red. Como todos ellos pueden cambiar rápidamente, el tiempo requerido para responder a una petición de usuario puede variar drásticamente de una petición a otra.

Ataques de los que deben defenderse los sistemas distribuidos

- Intercepción: Un atacante intercepta comunicaciones entre partes del sistema, provocando que haya poca confidencialidad.
- Interrupción: Los servicios son atacados y no pueden entregarse como se esperaba. Se bombardea al servicio con peticiones ilegítimas para que no pueda atender las legítimas.

- Modificación: 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 responda. Comunicación síncrona.
- 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 detalla los servicios requeridos por otro componente. Comunicación asíncrona.

Middleware
- Los componentes en un sistema distribuido pueden implementarse en diferentes lenguajes de programación y pueden ejecutarse en tipos de procesadores completamente diferentes. Los modelos de datos, la representación de la información y los protocolos de comunicación pueden ser todos diferentes. Un sistema distribuido requiere software que pueda gestionar estas partes distintas, y asegurarse de que dichas partes pueden comunicarse e intercambiar datos. Este software se sitúa en el medio de los diferentes componentes distribuidos del sistema.
- Por lo general, se implementa como un conjunto de librerías, que se instala en cada computadora distribuida.
- Normalmente, es de propósito general (enlatado), en lugar de sw hecho a medida. Ejemplos: comunicación con base de datos, gestores de transacciones, convertidores de datos, y controladores de comunicación.
- En un sistema distribuido brinda dos tipos de soporte:
  - Soporte de interacción → coordina las interacciones entre diferentes componentes del sistema. Puede soportar conversiones de parámetros si se usan lenguajes de programación diferentes.
  - Provisión de servicios comunes → proporciona implementaciones de reutilización de servicios que pueden requerir varios componentes en el sistema distribuido. Ejemplos: servicios de seguridad (autenticación y autorización), servicios de notificación y nomenclatura, servicios de gestión de transacción.

Evolución de arquitecturas de sistemas distribuidos
- Arquitectura maestro esclavo
  - Es el modelo más simple de un sistema distribuido, es un sistema multiprocesador en donde el sistema está formado por múltiples procesos que pueden ejecutarse sobre procesadores diferentes.
  - Se usa en sistemas de tiempo real en 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.
  - Ejemplo: sistema de control de tráfico. Un conjunto de sensores distribuidos recoge información sobre el flujo de tráfico y la procesan localmente antes de enviarla a una sala de control. Los operadores toman decisiones usando esta información y dan instrucciones a un proceso de control de diversas luces de tráfico.
- Arquitectura cliente servidor y arquitectura cliente servidor n-tier
- Arquitectura 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. Este tipo de sistemas está diseñado para sacar ventaja del poder computacional y de almacenamiento disponible en una red de computadores potencialmente enorme. Aquí cada nodo tiene la habilidad de comportarse como un cliente o un

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.
  - Forma parte de la vista de distribución.
  - Tiene alta disponibilidad.
  - Es altamente escalable y extensible.
  - Aprovechan la capacidad de procesamiento no utilizada de computadoras locales.
  - Muchos nodos pueden procesar la misma búsqueda.
  - Excesiva carga de trabajo al replicar comunicaciones entre pares.
  - Preocupaciones por la seguridad y la confianza.
  - Se usa cuando el sistema es de cómputo intensivo y se puede separar el procesamiento requerido en un gran número de cálculos independientes.
  - Cuando el sistema requiere intercambio de información entre computadoras individuales en una red y no hay necesidad de que esta información se almacene o gestione de manera centralizada.
  - Algunos ejemplos de estas arquitecturas son: bittorrent, napster.
- ❖ Arquitectura p2p descentralizada
  - Los nodos en la red no son simplemente elementos funcionales, sino que también son interruptores de comunicaciones que pueden encaminar los datos y señales de control de un nodo a otro.
  - Todos los componentes están en el mismo nivel jerárquico y ninguno controla a otro.
  - Cada componente de software tiene un rol dual, puede actuar como cliente o como servidor.
  - Surgen para atender muchas peticiones a la vez.
  - Ventaja → es altamente redundante, y por lo tanto es tolerante a fallos y tolerante a nodos desconectados de la red. Un cliente hace una petición, y el primero que esté disponible o que la tome, atiende esa petición.
  - Desventaja → existen sobrecargas obvias en el sistema ya que la misma búsqueda puede ser procesada por muchos nodos diferentes y hay una sobrecarga significativa en comunicaciones entre iguales replicadas. Se disminuye la performance.
- ❖ Arquitectura p2p semicentralizada
  - Hay un nodo superar, y el papel que cumple es ayudar a establecer contacto entre iguales en la red o para coordinar los resultados de un cálculo, es decir, asigna peticiones para evitar el procesamiento redundante.
  - El superpar debe conocer a todos los otros nodos.
  - Los nodos de la red se comunican con el servidor, para encontrar qué otros nodos se encuentran disponibles. Una vez que estos son encontrados, se pueden establecer comunicaciones directas y la conexión con el servidor es innecesaria.
  - Desventaja → el superpar puede saturarse y puede haber cuellos de botella en el mismo.
- ❖ Arquitectura espejada
  - En este estilo se duplica hardware idéntico que corre en paralelo.
  - Si uno de los servidores falla, el otro puede continuar por su cuenta. Se realiza la misma operación varias veces en paralelo.
  - El sw puede actualizarse en forma separada uno de otro.
  - La disponibilidad aumenta porque la probabilidad de que ambas fallen al mismo tiempo es baja.
- ❖ Arquitectura Rock
  - Los servidores se acomodan en pilas para utilizar menos espacio de piso.
  - Todas las computadoras de 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 muy rápida o la restricción de ancho de banda es menor, por lo que la comunicación entre computadoras del mismo rack es mucho más rápida.
- ❖ Arquitectura granja de servidores
  - Muchas computadoras son ubicadas en la misma habitación.
  - La granja puede componerse de varios racks.
  - Se las piensa como un recurso masivo que puede alojar cualquier aplicación.
  - Las aplicaciones tienen restricciones para poder ejecutarse en una granja, como no tener estado.

- Es fácilmente escalable agregando más hw del mismo tipo.
- Usos comunes: capa de interfaz de usuario y capa intermedia de un sistema de tres capas donde cada granja aloja una capa diferente.

MICROSERVICIOS

El sistema no se ejecuta como si fuera un único componente, sólo se ejecutan los componentes que son necesarios en ese momento. Cada componente se ejecuta en el momento en que son llamados o requeridos, y cada uno de ellos va a brindar un determinado servicio a través de sus interfaces.

Ayuda a mejorar la escalabilidad de una aplicación, pero disminuye la performance, ya que los componentes se encuentran distribuidos. No es autosuficiente. Es útil cuando el sistema es desarrollado por múltiples equipos distribuidos geográficamente; ayuda a trabajar con múltiples tecnologías.

Netflix trabaja con microservicios porque tienen múltiples equipos de trabajo que desarrollan una funcionalidad (servicio) particular, y seguramente lo hacen con una tecnología determinada.

El sistema está conformado por componentes independientes que brindan un servicio, y que se ejecutan, cada uno, por separado (posiblemente cada componente tenga su propia base de datos y su propia capa de presentación).

MONOLÍTICO

Es autosuficiente, se corre todo el sistema como un todo, no significa que no se deban respetar los principios de diseño y que los componentes no tengan que ser altamente cohesivos y débilmente acoplados.

Se suele pensar que se trata de un estilo anti patrón, pero no necesariamente es así, lo único que significa es que el sistema se ejecuta como si fuera un único componente, es decir, o se ejecuta todo o no se ejecuta nada.

No soporta escalabilidad, a medida que el sistema va creciendo, se hace cada vez más complicado de administrar, si el equipo es grande se complica el desarrollo de una manera eficiente.

Es mucho más sencillo de desarrollar; un ejemplo de sistema que es conveniente que sea monolítico son los sistemas ERP, ya que es conveniente que ese tipo de sistemas se ejecuten como un todo y que sean autosuficientes, y no dependan de la conexión a internet para seguir funcionando, pero todo eso va a depender de los requerimientos no funcionales que tenga el sistema y de la escalabilidad; si el sistema está pensando para que tenga un crecimiento rápido, la arquitectura monolítica no es la mejor opción.

ESTRATEGIA DE ENSAMBLAJE DE COMPONENTES

La ingeniería de software basada en componentes es una forma efectiva orientada a la reutilización para desarrollar nuevos sistemas empresariales. Se basa en la reutilización de componentes de software.

Es el proceso de definir, implementar e integrar componentes independientes y débilmente acoplados en los sistemas. Es una manera de entregar software complejo mucho más rápido.

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

La estrategia de ensamblado de componentes es una decisión arquitectónica y de diseño de la solución final del software a construir, que implica desde decidir implementar por componentes, definir la granularidad del componente hasta su ensamblado final y prueba de integración.

Es un modelo evolutivo de desarrollo de software. El modelo de ensamblaje de componentes incorpora muchas características del modelo en espiral.

Beneficios:

- Reducción de un 70% de tiempo del ciclo de desarrollo.
- Reducción de un 84% del coste del proyecto.
- Índice de productividad de 26.2 comparado con la norma de industria de 16.9.
- Estos resultados dependen de la robustez de la biblioteca de componentes.

Fundamentos:

- *Componentes independientes que son completamente especificados por sus interfaces:* debería haber una clara separación entre la interfaz de los componentes y su implementación, para que una implementación de un componente pueda reemplazarse por otra sin cambiar el sistema.
- *Estándares de componentes que facilitan la integración de estos:* estos estándares definen en el nivel más bajo cómo deberían especificarse las interfaces y cómo se comunican los componentes. Si los componentes cumplen con los estándares, entonces su funcionamiento es independiente del lenguaje de programación, por lo tanto, los componentes escritos en diferentes lenguajes pueden integrarse en el mismo sistema.
- *Middleware que brinda soporte de software para la integración de componentes:* para hacer que componentes independientes distribuidos trabajen en conjunto, es necesario dar soporte de middleware que maneje las comunicaciones de componentes. El middleware para el soporte de componentes maneja eficientemente los conflictos de bajo nivel y permite enfocarse en problemas relacionados con la aplicación. Además, un producto middleware, usado en este caso, puede proporcionar soporte para asignación de recursos, gestión de transacciones, seguridad y concurrencia.
- *Un proceso de desarrollo que se adapta a 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:

- Los componentes son independientes para que no interfieran su funcionamiento unos con otros.
- Se ocultan los detalles de la implementación, por lo que la implementación puede cambiar sin afectar al resto del sistema.
- Los componentes se comunican a través de interfaces bien definidas, de forma que, si estas interfaces se mantienen, un componente puede reemplazarse por otro que proporciona una funcionalidad adicional o mejorada.
- Las infraestructuras de componentes ofrecen varios servicios estándar que pueden usarse en sistemas de aplicación. Esto reduce los costos de desarrollo de aplicaciones.

Problemas:

- *Confiabilidad de los componentes*: los componentes son cajas negras y el código puede no estar disponible para los usuarios y no se puede garantizar la confiabilidad del componente, dado que puede no tener documentado los modos de fallo que comprometen al sistema si se usa ese componente, su comportamiento no funcional puede no ser el esperado, y podría llegar a tener código que comprometa la seguridad del sistema.
- *Certificación de componentes*: se debería certificar los componentes para asegurar su confiabilidad, pero ¿Quién pagará su certificación? ¿Quién será responsable si el componente no funciona de acuerdo a la certificación?
- *Predicción de componentes emergentes*: al ser componentes caja negra, predecir sus propiedades emergentes es difícil. Por eso, cuando se integran componentes, el sistema podría tener propiedades no deseadas que limitan su uso.
- *Equilibrio de requerimientos*: se tiene que encontrar el equilibrio entre requerimientos ideales y componentes disponibles en el proceso de especificación y diseño del sistema. Se necesita de un modelo de análisis de equilibrio sistemático y más estructurado que ayude a seleccionar y configurar componentes.

Un componente es una unidad de software independiente que puede estar compuesta por otros componentes, y que se utiliza para crear un sistema de software.

Un componente es un proveedor de servicios independiente. Cuando un sistema necesita de un servicio, llama a un componente que proporciona dicho servicio, sin tener en cuenta dónde se está ejecutando el componente o qué lenguaje de programación se ha utilizado para desarrollar el componente.

La visión de un componente como proveedor de servicios, resalta dos características críticas de componentes reutilizable:

- El componente es una entidad ejecutable independiente que puede estar formado por otros componentes ejecutables. No necesita ser compilado y puede usarse con otros componentes.
- Los servicios ofrecidos por un componente están disponibles a través de una interfaz, y todas las interacciones son a través de esa interfaz. La interfaz de componente se expresa en términos de operaciones parametrizadas, y su estado interno nunca se muestra.

Características de los componentes:

- Estandarizado: debe ajustarse a un estándar de componentes, que puede definir interfaces, metadatos, documentación, implementación y composición.
- Independiente: debe ser posible componerlo e implementarlo sin tener que usar otros componentes.
- Componible: todas las interacciones externas deben darse a través de interfaces públicamente definidas. Debe permitir acceso externo a información sobre sí mismo.
- Implementable: un componente debe ser independiente y debe ser capaz de funcionar como una entidad autónoma, o que funcione sobre una plataforma de componentes que implemente el modelo de componentes. Significa que no debe compilarse antes de ser implementado.
- Documentado: se debe documentar y especificar la sintaxis, semántica de todas las interfaces de componentes para que los usuarios puedan decidir si el componente satisface sus necesidades.
- Los componentes se definen por sus interfaces y puede considerarse que tiene dos interfaces relacionadas:
  - interfaz proporciona → define los servicios que ofrece el componente. Define los métodos que pueden ser llamados por los usuarios.
  - interfaz requiere → especifica qué servicios deben ser proporcionados por otros componentes en el sistema.

Modelo de componentes

Es una definición de estándares para implementación, documentación y despliegue. Se establecen con la finalidad de que los desarrolladores de componentes se aseguren de que estos puedan interoperar.

Los elementos fundamentales pueden clasificarse como:

- Elementos relacionados con las interfaces: Las interfaces son el elemento que define a un componente. El modelo de componentes especifica cómo deberían definirse sus interfaces, cómo debería ser el nombre de las operaciones, parámetros y excepciones que se incluyen en la definición de una interfaz. También debe especificar el lenguaje utilizado para definir las interfaces.
- Elementos relacionados con la información necesaria para utilizar el componente: Los componentes deben tener un nombre o manejador único asociado para que puedan distribuirse y ser accedidos remotamente. Los metadatos del componente son datos sobre el mismo componente, como información de atributos e interfaces. Es necesario para que los usuarios puedan saber qué servicios se proporcionan y cuáles se requieren.
- Elementos relacionados con la implementación: Los componentes son genéricos y deben ser configurados cuando se implementan a un entorno de aplicación particular. Se debe definir cómo se deben empaquetar, para poder implementarlos como entidades ejecutables independientes. El modelo de componentes debe incluir reglas para definir cuándo y cómo se permite el reemplazo. También el modelo debe definir la documentación asociada.

Los modelos de componentes, además de estándares, también son la base para el middleware de sistemas que proporciona el soporte para los componentes ejecutables.

Soporte de middleware

Servicios de plataforma: Son servicios fundamentales que permiten a los componentes comunicarse entre sí e interoperar en un entorno distribuido. Ejemplo: CORBA.

Servicios de soporte o servicios horizontales: Son servicios comunes que probablemente requieren ser usados por muchos componentes diversos. La disponibilidad de estos servicios reduce los costos de desarrollo de componentes y evita incompatibilidades.

Para usar los servicios ofrecidos por la infraestructura de un modelo de componentes, los componentes se implementan en un contenedor estandarizado predefinido. (Un contenedor es un conjunto de interfaces utilizadas para acceder a las implementaciones de los servicios de soporte). La inclusión del componente en el contendor proporciona automáticamente el acceso a los servicios.

Procesos ISBC

Desarrollo para reutilización: Se ocupa del desarrollo de componentes o servicios que se utilizarán en otras aplicaciones. Por lo general implica la generalización de componentes existentes.

Desarrollo con reutilización: Proceso para desarrollar nuevas aplicaciones, utilizando los componentes y servicios existentes.

Estos procesos tienen diferentes objetos, por ende, distintas actividades.

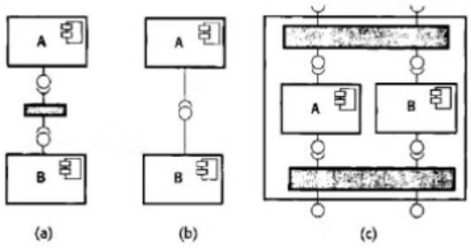
ISBC para reutilización – Desarrollo de componentes para reutilización

- Estos componentes pueden construirse generalizando componentes ya existentes.
- En primer lugar, hay que determinar si un componente realmente va a reutilizarse y, en segundo lugar, hay que evaluar si el ahorro que va a significar el componente reutilizado justifica el costo de hacer que este componente sea reutilizable.
- Para responder lo primero, debemos tener en cuenta:
  - Debería implementar un concepto del dominio que cambia muy lentamente. Ejemplo: en banco, las cuentas, los depósitos y las extracciones.
  - Todas las excepciones que se pueden producir deberían publicarse en la interfaz del componente.
  - Deberían esconder la representación del estado.
  - Deberían ser lo más independiente posible.
- Los cambios que pueden hacerse para que un componente sea más reutilizable incluyen:
  - Eliminar métodos específicos de aplicación.
  - Cambiar los nombres para hacerlos más generales.
  - Agregar métodos para brindar cobertura funcional.
  - Hacer manejadores de excepción consistente para todos los métodos.
  - Agregar una interfaz de configuración para permitir la adaptación de los componentes a diferentes situaciones de uso.
  - Integrar los componentes requeridos para aumentar la independencia.
- Debe existir un equilibrio entre reutilización y utilidad, ya que la reusabilidad añade complejidad, por eso dificulta su comprensibilidad.

ISBC con reutilización

- Bosquejar requerimientos del sistema → los requerimientos se esbozan, no se especifican con detalles.
- Identificar componentes candidatos → en base a los requerimientos esbozados, se identifican los componentes que pueden llegar a cumplir con estos.
- Modificar los requerimientos de acuerdo a los componentes descubiertos → se refinan los requerimientos en base a los componentes disponibles. Si los requerimientos no se pueden satisfacer completamente, se discute con los usuarios, que son flexibles a cambios de ideas, para que les entreguen un software más rápido y económico.
- Diseño arquitectónico → se diseña la arquitectura de acuerdo a los componentes identificados.
- Identificar componentes candidatos → se vuelve a repetir esta actividad dado que puede que algunos componentes puedan resultar no adecuados o pueden no trabajar adecuadamente con otros componentes seleccionados. Esto implica que puede haber cambios en los requerimientos también.

- Componer componentes para crear el sistema → es el proceso de desarrollo donde se integran los componentes con la infraestructura del modelo de componentes. A veces se requiere escribir código pegamento para integrar las interfaces de los componentes incompatibles.
- Características:
  - La etapa de diseño arquitectónico es muy importante dado que, se define el modelo de componentes a utilizar, se establece la organización de alto nivel y se toman decisiones sobre la distribución y control del sistema. Mientras más robusta sea la arquitectura mejor será la reutilización.
  - La identificación de componentes implica tres subactividades:
    - Búsqueda de componentes → se puede buscar entre componentes disponibles localmente o que sean de proveedores de confianza.
    - Selección de componentes → a partir de la lista de búsqueda se seleccionan los componentes que cumplan con los requerimientos. Se podría requerir decidir qué grupo de componentes nos dan el mejor cumplimiento de los requerimientos.
    - Validación de componentes → sirve para comprobar que los componentes se comportan como deberían hacerlo. Si el componente proviene de una fuente de confianza, la prueba individual no es necesaria y se lo prueba directamente cuando se lo integra con otros. Si viene de una fuente desconocida siempre se debe probar antes de incluirlo en el sistema. Implica desarrollar un conjunto de casos de prueba y desarrollar software adicional de pruebas para ejecutar esas pruebas. Se pueden presentar dos problemas: documentación del componente que no se encuentra lo suficientemente detallada como para escribir los casos de prueba, además puede haber funciones no deseadas que interfieran con el uso del componente pudiendo ocasionar fallas graves en el sistema.
- Composición de componentes
  - Es el proceso de ensamblar componentes unos con otros y con código pegamento especialmente escrito para crear un sistema u otro componente.
  - Tipos de composición:
    - Composición secuencial → tiene lugar cuando los componentes que constituyen un componente compuesto se ejecutan en secuencia. Se requiere código pegamento para poder 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 esperadas por el componente B.
    - Composición jerárquica → un componente llama a los servicios de otro. La interfaz provista de un componente está compuesta con la interfaz que requiere de otro. No aplica a servicios web que no tiene interfaz requerida.
    - Composición aditiva → 2 componentes se usan para crear un componente nuevo. No son dependientes ni se llaman mutuamente. Este tipo de componentes puede usarse con componentes que son unidades de programa o con componentes que son servicios. Se unen todas las interfaces de los componentes constituyentes eliminando las operaciones duplicadas si es necesario.



- Incompatibilidad de interfaces
  - Incompatibilidad de parámetros: Las operaciones de cada lado de la interfaz tienen el mismo nombre, pero sus tipos de parámetros o el número de parámetros son diferentes.
  - Incompatibilidad de operación: Los nombres de las operaciones en las interfaces que 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.
- Pueden aparecer conflictos entre requerimientos funcionales y no funcionales, y conflictos entre la necesidad de entrega rápida y la evolución del sistema. Será necesario decidir cuestiones tales como:
  - Qué composición de componentes es más efectiva para cumplir con los requerimientos funcionales.
  - Qué composición de componentes facilitará cambios a futuro.

- Cuáles serán las propiedades emergentes del sistema compuesto. Se trata de propiedades como rendimiento y confiabilidad. Sólo se podrán valorar después de implementar el sistema completo.

UNIDAD 4

EXPERIENCIA DE USUARIO – UX

La función de los profesionales de la experiencia de usuario es hacer que una tecnología sea amigable, satisfactoria, fácil de usar y realmente útil.

UX se refiere a cómo funciona realmente el producto, lo fácil o difícil que es de usar. UI se refiere a cómo se ve, si se ve agradable o no. Ambos aspectos son importantes, pero una buena experiencia de usuario prioriza la funcionalidad y comodidad sobre el diseño estético si es necesario (mas allá de que es cierto que los usuarios suelen preferir los diseños 'lindos', pero porque perciben que la usabilidad de estos es mejor, es decir, son más fáciles de usar).

- Accesibilidad: Es un atributo del producto que se refiere a qué tan posible es que éste sea usado por la mayor cantidad de personas posibles sin problema.
- Usabilidad vs Utilidad: Una mide la experiencia de usuario, cómo se siente el usuario al utilizar el sistema, y si puede cumplir sus objetivos de manera eficiente; mientras que el otro simplemente verifica que se cumplan los requerimientos. A qué le das más importancia va a depender del sistema que estés desarrollando y para quién es.
  La usabilidad y la utilidad son atributos que tienen mutua dependencia. Un producto será usable en la medida que el beneficio de usarlo justifique el esfuerzo necesario.
  Atributos de usabilidad:
  - Aprendizaje → cuánto tiempo tarda el usuario en aprender a utilizar el sistema y ser productivo.
  - Velocidad de funcionamiento → cómo responde el sistema a las acciones de trabajo que realiza el usuario sobre él.
  - Robustez → qué tolerancia tiene el sistema a los errores del usuario.
  - Recuperación → cómo se recupera el sistema a los errores del usuario.
  - Adaptación → qué tan atado está el sistema a una determinada forma de trabajar o modelo de trabajo; sí el modelo de trabajo cambia, tengo que cambiar el sistema o se puede adaptar a una nueva forma de trabajo o modelo de trabajo, por ejemplo, si de repente el modelo de trabajo en lugar de ser presencial pasa a ser virtual, puedo utilizar igual el sistema o lo tengo que modificar para que funcione, y sí lo tengo que modificar, qué tanto lo tengo que modificar.
- Arquitectura de la información: Es el arte, la ciencia y la práctica de diseñar espacios interactivos comprensibles, que ofrezcan una experiencia de uso satisfactoria, facilitando el encuentro entre las necesidades de los usuarios y los contenidos y/o funcionalidades del producto.
  Es un arte porque necesitamos creatividad para presentar la información de forma intuitiva y atractiva para el usuario.
  La ciencia porque tenemos que comprender cómo las personas procesan la información, el análisis de patrones de comportamiento y la aplicación de principios psicológicos.
  La práctica implica aplicar esos conocimientos para optimizar la disposición de elementos, la jerarquía de contenidos y la navegación, de manera fácil de usar y entender.
  Una buena *arquitectura de información* tiene en cuenta como los usuarios buscan, encuentran y entienden la información, lo que influye directamente en su satisfacción y efectividad al interactuar con su interfaz.
- Modelos mentales: Para diseñar productos usables y satisfactorios, nuestra primera misión es adquirir o construir un modelo de interacción preciso y completo, comprender cómo y con qué fines los usuarios utilizarán el producto, para de esta manera diseñar una interfaz adaptada al modelo mental de sus usuarios, y no una interfaz reflejo de nuestro propio modelo mental.

Estilos de interacción

Estilos	Ventajas	Desventajas	Ejemplos
Manipulación directa	-Interacción rápida e intuitiva. -Fácil de aprender.	-Puede ser difícil de implementar. -Adecuada sólo cuando hay metáfora visual para tarea y objeto.	-Videojuegos -Sistemas CAD
Selección de menús	-Evita errores del usuario. -Requiere teclear poco.	-Lenta para usuarios experimentados. -Compleja si existen muchas opciones.	-La mayoría de los sistemas de propósito general.
Rellenado de formularios	-Introducción de datos sencilla.	-Ocupa mucho espacio. -Problemas cuando las opciones del usuario no se ajustan a los campos.	-La mayoría de los sistemas transaccionales.
Lenguaje de comandos	-Poderos y flexible.	-Difícil de aprender. -Gestión pobre de errores.	-Sistemas operativos. -Sistemas de comandos y control.
Lenguaje natural	-Accesible a usuarios casuales. -Fácil de ampliar.	-Requiere teclear más. -No suelen ser fiables.	-Sistemas de recuperación de información.

PATRONES JENIFER TIDWELL

Hacen las cosas más fáciles de entender o más 'lindas', hacen a las herramientas más útiles y utilizables.

Los patrones pueden ser una descripción de buenas prácticas dentro de un dominio determinado.

Patrones de conducta

- Exploración segura: Déjame explorar sin perderme ni meterme en problemas.
- Gratificación instantánea: Quiero lograr algo ahora, no más tarde.
- Satisfacción: Esto es lo suficientemente bueno. No quiero pasar más tiempo aprendiendo a hacerlo mejor.
- Cambios en la mitad de la misión: Cambié de opinión sobre lo que estaba haciendo.
- Diferentes opciones: No quiero responder eso ahora, solo déjame terminar.
- Incremento gradual: Déjame cambiar esto. Esto no se ve bien, déjame cambiarlo de nuevo. Eso es mejor.
- Habituación: Esto funciona bien en otros lados ¿por qué no funciona bien también acá?
- Memoria espacial: Juro que este botón estaba aquí hace un minuto ¿a dónde se fue?
- Memoria prospectiva: Pongo esto aquí para recordarme a mí mismo tratarlo más tarde.
- Repetición simplificada: Tengo que repetir esto ¿cuántas veces?
- Teclado únicamente: Por favor, no me hagas usar el mouse.
- Consejos de otras personas: ¿qué dijeron los demás sobre esto?

Patrones de estructura física
Patrones de navegación
Patrones de disposición de elementos en la página
Patrones de acciones y comandos
Patrones para formularios y controles

Prototipos

Sirven para probar la experiencia de usuario, mejorarla y que el sistema final tenga una buena experiencia de usuario.

Se pueden diseñar prototipos en papel que sirvan simplemente para validar colores, formas, ubicación de íconos.

El problema con los prototipos es que le tiene que quedar claro al usuario que no es el sistema funcional. Y el otro problema es que los desarrolladores también tienen que entender que éstos no siempre van a ser evolutivos, la gran mayoría de las veces van a ser desechables, por esto es que antes de empezar a definir la funcionalidad y desarrollar un prototipo hay que determinar cuál es su objetivo, luego diseñarlo y desarrollarlo y por último probarlo.

El uso de prototipos es clave para evaluar la interacción, retroalimentar y mejorar. La retroalimentación tiene que ser temprana para corregir lo antes posible y gastar menos, porque mientras más te tardas en corregir los errores, más avanzaste en una dirección que era equivocada.

Hay que realizar ciclos cortos porque la información llega de forma gradual y hay que empezarla a tratar lo antes posible.

¿Por qué necesitamos un diseñador de interfaces o un diseñador de interacción?

Producen interfaces con menores errores, las interfaces que diseñan permiten una ejecución más rápida, entienden mejor al usuario (modelo mental) que los desarrolladores.

¿Por qué es tan complicado el diseño de interfaces de usuario?

Dificultad por parte de los diseñadores para comprender las tareas del usuario, especificaciones incompletas o ambiguas, las interfaces deben satisfacer las necesidades, experiencias y expectativas de los usuarios previstos, amplia diversidad de usuarios con diferentes características, los programadores tienen dificultades en pensar como los usuarios, complejidad inherente de las tareas y los dominios, requerimientos específicos del dominio

Proceso de diseño de interfaz

Todo necesita de un proceso para llevarse a cabo. El 63% de los proyectos se pasan del presupuesto inicial porque:

- los usuarios piden cambios.
- tareas no tomadas en cuenta.
- los usuarios no entendían sus propios requerimientos.
- comunicación y entendimiento insuficiente entre el usuario y el desarrollador.

El proceso de diseño de interfaz también es iterativo e incremental. Necesitas retroalimentación constante por parte del usuario. Para poder hacer una primera evaluación se utilizan prototipos desechables. Puede que se tengan que hacer varios diseños hasta que el usuario se quede con uno. Es en ese momento cuando empezamos a diseñar prototipos evolutivos. Los prototipos evolutivos sirven para probar la interacción del usuario con el sistema. Una vez que los prototipos han pasado las pruebas de usabilidad, allí se genera una versión beta o alfa que se puede poner en producción.

8 reglas de oro

1. **Buscar siempre la coherencia**: Relacionado a la heurística de Nielsen de mantener la consistencia y estándares. Busca que se mantenga siempre el mismo estilo de presentación para todo el sistema, misma ubicación de botones, colores, etc.
2. **Permitir el uso de atajos**: Relacionado a la heurística de Nielsen de flexibilidad y eficiencia en el uso.
3. **Dar retroalimentación de información**: Relacionado a la heurística de Nielsen de visibilidad del estado del sistema; por cada acción que realiza un usuario en el sistema, debería existir una respuesta que sea visible por parte del sistema en tiempo adecuado (ruedita de carga, por ejemplo).
4. **Diseñar diálogos que tengan un fin**: Relacionado a la heurística de Nielsen de diseño estético y minimalista; no se deben incluir cosas que no sean necesarias y que no agreguen valor al usuario (por ejemplo, para una interfaz de compra de boletos sólo necesito saber el precio de los boletos para un determinado origen y destino que yo seleccione).

5. **Permitir el manejo simple de errores**: Relacionado con la heurística de Nielsen de ayudar a los usuarios a reconocer, diagnosticar y recuperarse de los errores; cuando ocurra un error mostrarle al usuario cómo recuperarse de él y que pueda seguir utilizando el sistema con normalidad.
6. **Permitir deshacer las acciones con facilidad**: Se relacionada con la heurística de Nielsen de libertad y control por parte del usuario; permitirle al usuario volver atrás y arrepentirse de algo que hizo.
7. **Permitir que el centro de control sea interno**: Se relacionada con la heurística de Nielsen de visibilidad y estado del sistema, libertad y control por parte del usuario y prevención de errores; una interfaz debe hacerle creer al usuario que él tiene el control.
8. **Reducir la carga de memoria inmediata**: Se relaciona a la heurística de Nielsen de reconocer antes que recordar; no hacerle recordar al usuario lo que vio en la pantalla anterior, por ejemplo, si estoy comprando algo por internet no necesito irme a la pantalla anterior o estar recordando el precio de la compra, se ve constantemente a lo largo de todas las pantallas.

Heurísticas de Nielsen

I. **Visibilidad del estado del sistema**: Mostrarle al usuario en qué estado se encuentra el sistema, por ejemplo, si se está cargando un archivo, que se muestre en pantalla que se está realizando la carga y en qué porcentaje de carga se encuentra.
II. **Documentación y ayuda**: Los sistemas deben tener un lugar de documentación que aclare dudas sobre el producto, por ejemplo, ícono de información que muestra cómo funciona el sistema o cómo usar una determinada funcionalidad.
III. **Libertad y control por parte del usuario**: Permitir a los usuarios volver atrás.
IV. **Prevenir errores**: Cuadros de confirmación antes de realizar una acción destructiva; tratar de frenar a los usuarios antes de que cometan un error.
V. **Ayudar a los usuarios a reconocer, diagnosticar y recuperarse de los errores**: Mostrar mensajes de error que sean claros y que ayuden a los usuarios a entender cuál es el problema y qué es lo que tiene que hacer para recuperarse de ese error.
VI. **Relación entre el sistema y el mundo real**: Usar los términos que se utilizan en el negocio, es decir, el sistema debe utilizar las palabras que se utilizan en el ambiente en el que se va a utilizar el sistema.
VII. **Consistencia y estándares**: Si por ejemplo definiste que los botones de confirmación van a estar a la derecha y van a ser de color verde, mantenerlo de esa manera para todo el sistema.
VIII. **Reconocer antes que recordar**: Evitar mucho texto, utilizar imágenes o íconos que el usuario pueda asociar a una determinada acción. Por ejemplo: la lupita para buscar información.
IX. **Flexibilidad y eficiencia en el uso**: Permitir el uso de atajos para usuarios que son más avanzados.
X. **Diseño estético y minimalista**: Menos, es más; se debe visualizar la información justa y necesaria, no se debe agregar información o datos de más que, muchas veces, no son útiles y que lo único que hacen es confundir más a los usuarios.

UNIDAD 5

EVOLUCIÓN DEL SOFTWARE

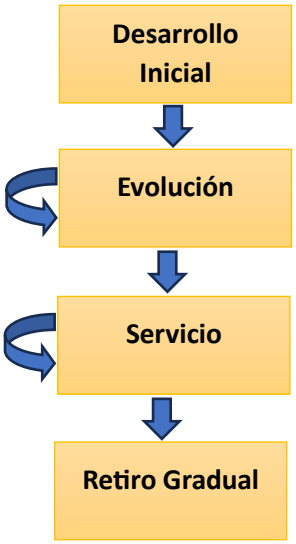
El software siempre va a evolucionar con el tiempo porque vivimos en un mundo que cambia constantemente. Es el cambio quien impulsó el proceso de evolución del software.

Para el software, el cambio ocurre cuando se corrigen errores, cuando el sistema se adapta a un nuevo entorno, cuando el cliente solicita nuevas características o funciones, y cuando la aplicación se somete a reingeniería para ofrecer beneficios en un contexto moderno.

Después de distribuir un sistema, inevitablemente debe modificarse de manera constante, con la finalidad de mantenerlo útil y que no se vuelva obsoleto.

Una vez construido el software y liberado la primera versión de nuestro sistema, hay que empezar a mantener y evolucionar el mismo.

EL CICLO DE VIDA EN LA EVOLUCIÓN DEL SOFTWARE



Es lo que se construye antes de poner en producción la primera versión del sw. Definimos un proyecto y cuando se termina, se lo pone a producción y pasa a la siguiente etapa.

Puesta en producción de la primera versión. Se ingresan nuevos requerimientos, se corrigen errores.

El software ya está en uso; decidimos no incorporar más requerimientos, sólo se hace corrección de errores, le damos mantenimiento.

Cuando no le damos más mantenimiento al software; se lo reemplaza por otro o se lo retira porque el negocio ya no lo necesita.

Identificación de cambio y procesos de evolución

Aceptar las peticiones de cambio de los clientes y determinar cómo las vamos a tratar. Se necesita incorporar herramientas que nos permitan receptar peticiones de cambio o correcciones. El más usado actualmente es JIRA.

Los procesos de evolución dependen de la organización, del tipo de software, de los procesos de desarrollo usados en la organización y de las habilidades de las personas que intervienen.

La evolución en algunas organizaciones puede ser un proceso informal, donde las solicitudes de cambio provienen de conversaciones entre los usuarios del sistema y los desarrolladores. Y en otras puede ser un proceso formalizado con documentación estructurada generada en cada etapa del proceso.

Las propuestas de cambio son el motor para la evolución de un 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 reporte 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 lo largo del ciclo de vida del producto.

El *proceso de evolución* contiene las siguientes actividades fundamentales:

- Análisis de impacto→significa analizar lo que implica ese cambio a nivel de esfuerzo que va a llevar, cuántos componentes va a afectar y qué hay que cambiar. Se realiza antes de decir si ese cambio se va a llevar a cabo; por esto es importante diseñar para el cambio siguiendo los principios de diseño y aplicando los patrones que aplican los distintos principios (acá es donde se ve la calidad del diseño).
- Planificación de versiones→consiste en tres subetapas: reparación de fallas, adaptación (en el caso de que existe algo nuevo en el ambiente, como un SO nuevo) y perfeccionamiento del sistema (nueva funcionalidad).
- Implementación de cambios→se deben llevar a cabo los wf de análisis, diseño, implementación y prueba. Se define cuando se va a realizar en función de las propiedades.
- Liberación del sistema a los clientes→se lanza una nueva versión del sistema.

El costo y el impacto de dichos cambios se valoran para saber qué 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 de cuáles cambios implementar en la siguiente versión del sistema, según las prioridades definidas. Después de implementarse

se valida y se libera una nueva versión del sistema. Luego el proceso se repite con un conjunto nuevo de cambios propuestos para la siguiente versión.

Proceso de implementación de cambios: peticiones de cambio, análisis de requerimientos, actualización de requerimientos (actualizar la arquitectura y el resto de los modelos existentes para que sean consistentes con el sw que se encuentra en producción), desarrollo del software.

Proceso de reparación de fallas de emergencia: peticiones de cambio, análisis de código fuente, modificación de código fuente, entregar sistema modificado. Si la reparación es urgente entonces directamente se entra al código, se lo arregla, se lo entrega y luego de eso se debería ir pasos hacia atrás y actualizar los modelos y la arquitectura para mantenerlos consistentes con el sw. Generalmente esto no se hace, ya que se generan inconsistencias y se provoca que el esfuerzo de haber realizado dichos modelos se eche a perder, por eso realizar mantenimiento es tan costoso, porque la documentación generalmente no se encuentra actualizada.

Dinámica de evolución de programas – Leyes de Lehman

- Cambio continuo → un programa que es utilizado en un entorno del mundo real debe necesariamente cambiar o cada vez se volverá menos útil a ese entorno.
- Incremento de complejidad → como un programa en evolución cambia, su estructura en evolución tiende a ser cada vez más compleja. Deben asignarse recursos extra para preservar y simplificar esta estructura.
- Evolución de grandes programas → la evolución del programa es un proceso autorregulado. Atributos del sistema tales como tamaño, tiempo entre releases, y el número de reporte de errores, son invariantes, en general para cada release.
- Estabilidad organizacional → a través de la vida de un programa, su ratio de evolución es constante e independiente de los recursos dedicados al desarrollo del sistema; ritmo de evolución estable en el tiempo.
- Conservación de familiaridad → conforme un sistema evoluciona, también lo hace todo lo asociado a él.
- Crecimiento continuo → la funcionalidad ofrecida por los sistemas tiene que crecer continuamente para mantener la satisfacción del usuario.
- Disminución de calidad → la calidad de los sistemas disminuirá a menos que se modifiquen para reflejar los cambios en su entorno operativo.
- Retroalimentación de sistemas → la evolución de los procesos incorpora retroalimentación de sistemas multiagente, multiciclo, y hay que tratarlos como sistemas de retroalimentación para lograr una mejora significativa del producto.

La aplicabilidad de estas leyes ha sido probada para sistemas grandes hechos a medida, desarrollados para organizaciones grandes.

ESTRATEGIAS DE EVOLUCIÓN DEL SOFTWARE

Reingeniería

Significa reestructurar o reescribir el código de un software antiguo, una vez que ya fue mantenido. Es decir, se crea un nuevo software a partir de uno ya existente.

Significa mejorar la calidad interna de un producto de software, pero sin cambiar ninguna funcionalidad. Busca hacer que el producto sea más fácil de mantener. El sw hace lo mismo que antes, pero con una calidad mayor.

El propósito de hacer reingeniería es la necesidad de mantenimiento del sw, pero a un costo menor.

Para hacer que los sistemas heredados sean más fáciles de mantener se les puede aplicar reingeniería para mejorar su estructura interna y entendimiento.

La reingeniería puede implicar volver a documentar el sistema, hacer refactoring de la 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 sistema no cambia y normalmente es conveniente tratar de evitar grandes cambios en la arquitectura del sistema.

Ventajas de la reingeniería frente a la sustitución

- ✓ 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, hay un riesgo alto al realizar un software desde cero.
- ✓ Reducción de costos: El costo de reingeniería puede ser significativamente menor que el costo de desarrollar un software nuevo.

Actividades de la reingeniería

- ➢ Traducción del código fuente: El programa se convierte de un lenguaje de programación antiguo a una versión más moderna, o a un lenguaje diferente.
- ➢ Ingeniería inversa: Es el proceso de analizar un programa para crear una representación del mismo en un nivel más alto de abstracción que el código. Se realiza cuando los sistemas no cuentan con documentación actualizada o directamente no existe. Permite crear modelos a partir de código. No se toca el producto, sino que se está mejorando la calidad al generar los modelos que se deberían haber hecho antes, para que sea más fácil de mantener. Es un proceso de recuperación del diseño.
- ➢ 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.
- ➢ 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, se debe tener en cuenta la alta cohesión y el bajo acoplamiento.
- ➢ Reingeniería de datos: Significa la redefinición de los esquemas de bases de datos y convertir las bases de datos existentes a la nueva estructura.

No necesariamente se tienen que aplicar todas las actividades. Se debe tener en cuenta la trazabilidad existente entre los modelos para que sea menos costoso incluir cambios. Actualmente, la gran mayoría de las actividades de reingeniería pueden ser automatizadas.

Desventajas de la reingeniería

- 𝓧 No es posible convertir un sistema funcional en un sistema orientado a objetos.
- 𝓧 Grandes cambios no pueden hacerse de forma automatizada.
- 𝓧 Los sistemas con reingeniería probablemente no serán tan mantenibles como los sistemas nuevos desarrollados con técnicas modernas.

Factores del costo de reingeniería

- La calidad del software al que se le va a aplicar reingeniería.
- La herramienta de soporte disponible.
- La extensión de la conversión de datos que se requiere.
- La disponibilidad de personal experto para realizar la reingeniería.

Factores de complejidad

- complejidad de control → puede medirse examinándose las sentencias condicionales en el programa.
- complejidad de datos → la complejidad de las estructuras de datos y los componentes de interfaces
- tamaño de los módulos.
- comentarios del programa → tal vez más comentarios significan mantenimiento más fácil.
- extensión de los nombres identificadores → los nombres más largos implican mayor legibilidad.

Refactoring (proceso continuo de mantenimiento)

Se da cuando tenemos variables que cambian continuamente, es decir, es un proceso de cambio continuo. Se utiliza para evitar la degradación del software y para extenderle la vida útil.

Es un proceso de hacer mejoras a un programa para frenar la degradación producida por el cambio. Es decir, modificar un programa para mejorar su estructura, reducir su complejidad o hacerlo más fácil de entender.

Es una técnica para mejorar el código por dentro, pero sin modificar su estructura por fuera. Se parte de un componente de código que funciona, pero queremos modificarlo para mejorarlo y que sea más eficiente o simplemente que se lo entienda mejor. No se agrega funcionalidad, hay que centrarse en la mejora del programa.

Es una herramienta para mejorar o evolucionar la calidad interna de un componente, una vez que éste ha sido construido y funciona.

No es lo mismo que la reingeniería, aunque ambas tienen la intención de modificar el sw para aumentar su calidad.

Es una forma disciplinada de introducir cambios reduciendo a posibilidad de introducir defectos.

Los cambios en el refactoring: no modifican el comportamiento observable, eliminan la duplicación o la complejidad innecesaria, mejoran la calidad del software, hacen que el código sea más simple y más fácil de entender, flexibiliza el código, hace que el código sea más fácil de cambiar.

Para hacer refactoring se aplican los principios y patrones de diseño.

Reingeniería	Refactorización
Se lleva a cabo después de haber mantenido un sistema durante mucho tiempo. Se crea un nuevo sistema a partir de un sistema heredado utilizando herramientas automatizadas.	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 el sistema

Mejoras de la refactorización: evita el código duplicado, evita los métodos largos, evita las sentencias case (gracias al polimorfismo), evita la aglomeración de datos, elimina la generalidad especulativa.
¿Por qué es necesario hacer refactoring?: evita el deterioro del diseño, limpia el desorden en el código, simplifica el código, incrementa la legibilidad y comprensibilidad, facilita el encontrar errores, reduce el tiempo de depuración, incorpora el aprendizaje que hacemos sobre la aplicación, rehacer las cosas es fundamental en todo proceso creativo.

Refactoring en el proceso de desarrollo y evolución del software

- Extreme programming: es una metodología ágil. Es refactoring embebido en desarrollo conducido por testing (TDD).
- Refactoring y TDD: es una estrategia de desarrollo conducido por testing, se hace cuando se construye software por primera vez. Cuando se pasaron las pruebas, se entra al código y se mejora la calidad interna, pero haciendo que las pruebas sigan pasando. Luego se vuelve a probar.
- Refactoring como mantenimiento preventivo: sirve para mejorar la calidad y evitar la reingeniería.

Mantenimiento

Es el proceso general de modificar el software una vez entregado, para que siga siendo útil. Los cambios pueden ser la corrección de errores de codificación, diseño o especificación, incorporar/modificar requerimientos, modificar las instalaciones sin cambiar los RF, sino los RNF. Pueden modificar o agregar funcionalidad al sistema.

Puede requerirse por tres motivos: corregir fallas en el software; adaptar el software por una corrección de requerimientos; modificar el software a un nuevo entorno operativo.

El mantenimiento es un proceso que ocurre casi de forma inmediata, apenas el sw se libera los reportes de errores empiezan a llegar y nos enfrentamos a una lista a de corrección de errores, peticiones de adaptación y mejoras categóricas que deben planearse, calendarizarse y lograrse.

Tipos de mantenimiento:

- ❖ Correctivo: una vez entregado en software, éste presenta fallas o errores al momento de su utilización, en aspectos que no fueron tenidos en cuenta cuando se realizaron los tests. Es no cobrable.

- ❖ Adaptativo: cuando algún aspecto del entorno del sistema cambia el software (extendés el sistema, amplias la BD, incorporás componentes que ayuden con la rapidez, etc). Son modificaciones de los RNF. Es cobrable.
- ❖ Perfectivo: se da cuando el cliente desea agregar nuevas funcionalidades o cambian las existentes. Son cambios en los RF. Es cobrable.

Factores del costo de mantenimiento

- Estabilidad del equipo: la gente no quiere hacer mantenimiento porque no es una actividad desafiante ni que te permita crecer personalmente, ni es motivante. Se da mucho recambio de gente y es costoso que una persona nueva venga y modifique un sw que no conoce dado que la arquitectura no está documentada (generalmente), se tiene que ver código escrito por otro y lleva tiempo comprenderlo y si lleva tiempo, es más costoso.
- Prácticas de desarrollo deficientes.
- Habilidad del personal.
- Antigüedad y estructura del programa: mientras más viejo es el software, se degrada más, y más difícil es introducir cambios.

Predicción de cambios

Se puede hacer una proyección de futuros cambios analizando los requerimientos y viendo cuáles son más volátiles que otros, cuáles son más susceptibles a cambiar con el tiempo, analizando las decisiones que quedaron aferradas al ambiente o contexto del sistema.

Si esto cambia, el sistema también deberá hacerlo (ejemplo: leyes contables, bases imponibles de impuesto a las ganancias, topes de aportes, etc).

Hay que entender que cuando el software se pone en producción, ahí no termina nuestro trabajo, sino que recién comienza.

Para predecir cambios necesitamos entender la relación que tiene nuestro sistema con el ambiente externo. Los sistemas que se encuentran fuertemente acoplados a su ambiente son sistemas que siempre que el ambiente cambia, este obligatoriamente tiene que cambiar. Para evaluar la relación que existe entre un sistema y su ambiente, los factores que influyen son:

- El número y complejidad de las interfaces del sistema→cuantas más interfaces y más complejas sean, más probable es que se tengan que cambiar las interfaces a medida que se propongan nuevos requerimientos.
- El número de requerimientos volátiles del sistema→los requerimientos que reflejan procedimientos o políticas de la organización son más inestables que los requerimientos que se encuentran en características de un dominio estable.
- Los procesos de negocio en los que se usa el sistema→a medida que evolucionan los procesos de negocio, generan peticiones de cambio del sistema.

La aceptación de cambio depende de la mantenibilidad de los componentes afectados por el cambio. La implementación de los cambios degrada el sistema y reduce la mantenibilidad. Los costos de mantenimiento dependen del número de cambios y los costos del cambio.

Un software que se encuentra funcionando y que tenemos que evolucionar se le llama sistema heredado y puede estar en una de cuatro situaciones:

- o Calidad del sistema alta y valor de negocio alto: situación ideal a la que se debe apuntar. Se debe aplicar la estrategia de mantenimiento (eventualmente sustitución si se elevan los costos). Ejemplo, sistema de facturación en un supermercado.
- o Calidad del sistema alta y valor de negocio bajo: sistema de administración de clientes en un súper, en este caso se hace mantenimiento también.
- o Calidad del sistema baja y valor de negocio alto: es la peor situación. Es un sw muy importante que, al tener baja calidad, tiene alta demanda de mantenimiento. Se debe aplicar reingeniería para mejorar su calidad y pasarlo al cuadrante superior derecho (alta calidad y alto valor de negocio).
- o Calidad del sistema baja y valor de negocio bajo: se debe sustituir por algo mejor.