ResumenFinalDSI.pdf

user_3706713

Diseño de sistemas

3° Ingeniería en Sistemas de Información

Facultad Regional Córdoba
Universidad Tecnológica Nacional

Unidad Nro. 1: Análisis de Sistemas de Información Orientado a Objetos con UML

UML 2.0

Lenguaje Unificado de Modelado: Lenguaje estándar para describir planos de SW. Es sólo un lenguaje es sólo una parte de un método de desarrollo de SW, es independiente del proceso. Para utilizarlo óptimamente se debe usar en un proceso *dirigido por CU, centrado en la arquitectura, iterativo e incremental.*

UML es un lenguaje: Proporciona un *vocabulario y reglas* posibilitando la comunicación en el equipo de trabajo, que se centran en la *representación conceptual y física de un sistema*: indican cómo crear y leer modelos bien formados, pero no dicen qué modelos crear ni cuándo (eso hace el PUD). Nos ayuda a la compresión del sistema, a través de múltiples modelos conectados entre sí.

UML nos permite:

- ***Visualizar:*** Detrás de cada símbolo hay un significado bien definido. Así, un desarrollador puede escribir un modelo UML y otro desarrollador o herramienta, puede interpretarlo sin ambigüedad.
- ***Especificar:*** Construir modelos precisos, no ambiguos y completos.
- ***Construir:*** Te permite establecer *correspondencias* desde un modelo UML a un lenguaje de programación, permitiendo *ingeniería directa* (generación de código a partir de un modelo UML) e *ingeniería inversa* (reconstruir un modelo UML a partir de código).
- ***Documentar:*** Nos permite documentar la arquitectura de un sistema y sus detalles, como también los requerimientos, pruebas y modelar las actividades de planificación de proyectos y gestión de versiones.

Bloques básicos de construcción

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

Autor: Andermatten – Martinez – Sánchez Díaz

❖ ***Elementos***

1.1 ***Estructurales:*** partes estáticas de un modelo, representan conceptos o cosas materiales.

- *Clase:* Descripción de un conjunto de objetos que comparten atributos, operaciones, relaciones y semántica.
- *Interfaz:* Define un conjunto de operaciones (no su implementación) que especifican un servicio de una clase.

  Es un tipo de clase especial, es abstracta y sus métodos no contienen implementación
- *Colaboración:* Define una interacción (Comunicación entre objetos) y es una sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento mayor que la suma de los comportamientos de sus elementos.
- *Caso de Uso:* Describe un conjunto de secuencias de acciones que ejecuta un sistema y produce un resultado observable de interés para un actor particular.
- *Clase Activa:* Sus objetos tienen uno o más procesos o hilos de ejecución, con comportamiento concurrente.
- *Componente:* Parte modular del diseño del sistema que oculta su implementación y brinda su servicio a través de interfaces externas.
- *Artefacto:* Es una parte física y reemplazable de un sistema que contiene información física (bits).(Archivo fuente, Scripts, Ejecutables, etc)
- *Nodo:* Es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional, que suele tener memoria y capacidad de procesamiento.

1.2 ***De Comportamiento:*** partes dinámicas de los modelos de UML.

- *Interacción:* Comportamiento que comprende mensajes intercambiados entre objetos, dentro de un contexto particular, para alcanzar un objetivo específico.
- *Máquina de estados:* Comportamiento que especifica las secuencias de estados por las que pasa un objeto o una interacción durante su vida en respuesta a eventos, y sus reacciones a estos.
- *Actividad:* Comportamiento (método) que especifica la secuencia de pasos que ejecuta un proceso computacional.

1.3 ***De Agrupación:*** partes organizativas de los modelos UML.

- *Paquete:* Mecanismo de propósito general para organizar un modelo (elemento de agrupación básico).

1.4 ***De Anotación:*** partes explicativas de los modelos UML, comentarios que se pueden aplicar para describir, clasificar y hacer observaciones sobre cualquier elemento.

- *Nota:* Símbolo para mostrar restricciones y comentarios junto a elementos. Nos sirve para explicar lo que sea de un Modelo

❖ ***Relaciones:*** Comunicación entre los elementos

2.1 ***De Dependencia:*** relación *semántica* entre dos elementos, en la que un cambio a un elemento (independiente) puede afectar la semántica del otro (dependiente). (- - - - - >) Clases de fabricación pura (gestor con las clases). Clase A ----> Clase B (A depende de B)

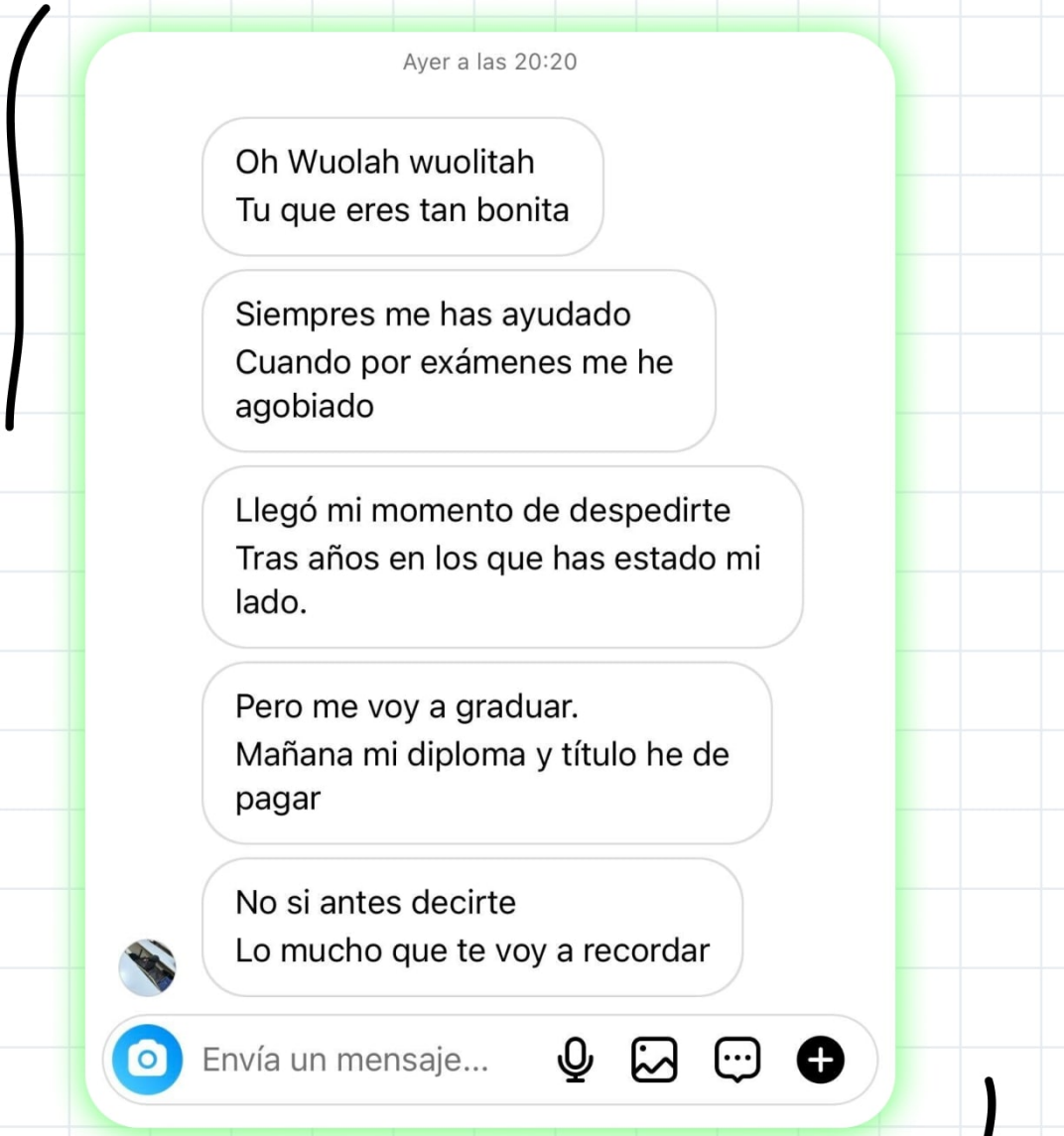
2.2 ***De Asociación:*** relación *estructural* entre clases que describe un conjunto de enlaces. (▯)

- *agregación:* tipo especial de asociación que representa una relación estructural entre un todo y sus partes. (rombo

Autor: Andermatten – Martinez – Sánchez Díaz

Que no te escriban poemas de amor cuando terminen la carrera 🚩🚩🚩🚩🚩🚩

(a nosotros por suerte nos pasa)



WUOLAH

vacío). Composición débil. Formado por un componente y un compuesto. Si se elimina el compuesto, no se elimina el componente.

- *composición:* Composición Fuerte. Formado por un componente y un compuesto. Si se elimina el compuesto, se **elimina** el componente. Dependen uno del otro (rombo lleno)

2.3 ***De Generalización/Herencia:*** relación *de especialización/generalización* en la que el elemento especializado (hijo) se basa en la especificación del generalizado (padre), compartiendo su estructura y comportamiento.

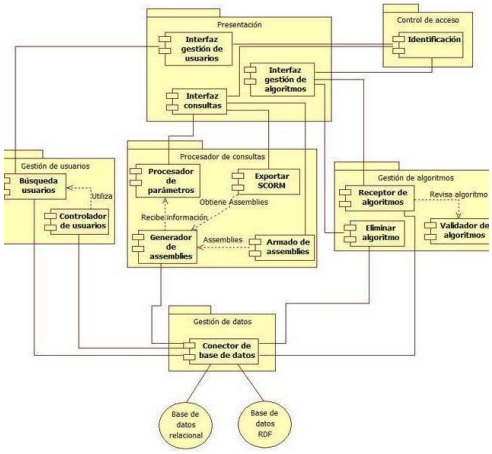
2.4 ***De Realización:*** relación *semántica* entre clasificadores, donde un clasificador especifica un **contrato** que otro garantiza que cumplirá.

❖ ***Diagrama:*** Es la representación gráfica de un conjunto de elementos, visualizado como un grafo conexo de nodos (*elementos*) y arcos (*relaciones*). Se dibujan para visualizar un sistema desde diferentes perspectivas, de forma que un diagrama *es una proyección de un sistema*. Se organizan en paquetes. Puede contener cualquier combinación de elementos y relaciones. *En la práctica surgen un número de combinaciones habituales, que son **consistentes con las cinco vistas que comprenden la arquitectura de un sistema**.*

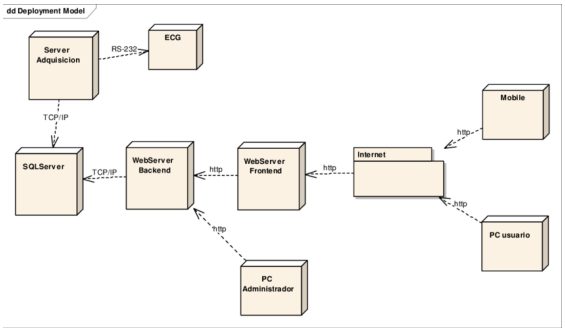
Se dividen en dos categorías:

◦ ***Estructurales:*** sirven para representar las *partes estáticas de un sistema.*

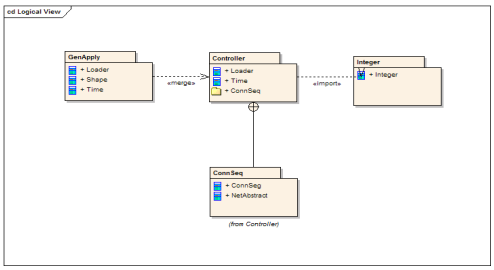
- *de Clases:* muestra un conjunto de clases, interfaces y colaboración, y sus relaciones.
- *de Objetos:* muestra un conjunto de objetos y sus relaciones. Representación estática de instancias de clases.
- *de Componentes:* representa la encapsulación de una clase, sus interfaces, puertos y estructura interna, la cual está formada por otros componentes anidados y conectores. *Vista de implementación estática del **diseño**.*



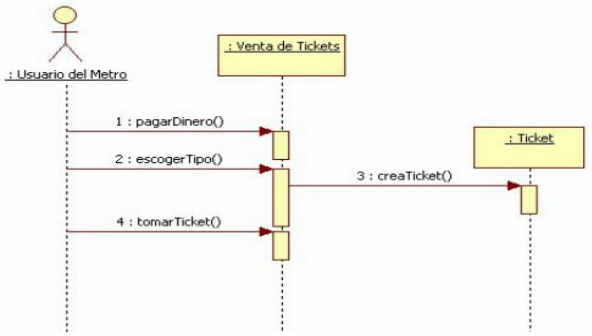
- *de Despliegue:* muestra la configuración de nodos de procesamiento en tiempo de ejecución y los artefactos que residen en ellos. *Vista de despliegue estática de una **arquitectura**.*



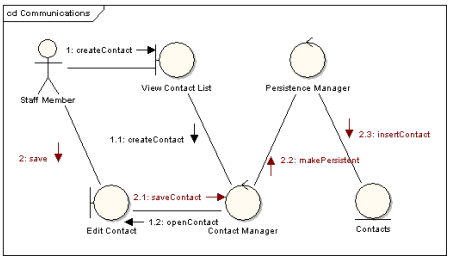
- *de Artefactos:* muestra los constituyentes físicos de un sistema en el computador (archivos, BD).
- *de Paquetes:* muestra la descomposición del modelo en unidades organizativas y sus dependencias.



- **Comportamiento:** sirven para representar las *partes dinámicas de un sistema*.
- *De Interacción:* muestra una interacción, entre un conjunto de objetos y los mensajes que se envían.
  - *De Secuencia:* resalta la *ordenación temporal de los mensajes*. Permite modelar la línea de vida de un objeto. Un mensaje se representa con una flecha de una línea de vida a otra.
    - *línea de vida:* es discontinua y vertical, representa la existencia de un objeto en un período de tiempo.
    - *foco de control:* rectángulo que representa el período de tiempo en el que un objeto ejecuta una acción.



  - *De Comunicación:* resalta la *organización estructural de los objetos* que envían y reciben mensajes. Permite modelar los enlaces estructurales entre los objetos de la interacción.
    - *camino:* línea que se dibuja haciéndola corresponder con una asociación.
    - *número de secuencia:* indica el orden del mensaje.



- *de tiempos (timing):* muestra los *tiempos reales entre diferentes objetos*. *Es a nivel de instancia (objeto)*.
- *de descripción de interacción*

- *De Estados:* muestra una máquina con estados, transiciones, eventos y actividades. Destaca los estados potenciales de los objetos y las transiciones entre esos. *Es a nivel de clase*. *Vista dinámica de un objeto.*
- *De Casos de Uso:* muestra un conjunto de CU y actores y sus relaciones. Organiza y modela el comportamiento del sistema.
- *De Actividades:* muestra la estructura de un proceso paso a paso. Destaca el flujo de control entre actividades.

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

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

*También hay modelos: **abreviados** (ciertos elementos se ocultan para simplificar la vista), **incompletos** (pueden estar ausentes ciertos elementos) e **inconsistentes** (no se garantiza la integridad del modelo).*

Mecanismos comunes de UML

- ❖ ***Especificaciones:*** detrás de cada elemento de su notación gráfica hay una especificación que proporciona una explicación textual de la sintaxis y semántica de ese bloque de construcción.
- ❖ ***Adornos:*** todos los elementos en la notación UML parten de un símbolo básico, al cual pueden añadirse adornos específicos de ese símbolo. Para atributos y métodos: *+ (public), - (private) y # (protected).*
- ❖ ***Divisiones comunes***
  - *Entre clase y objeto:* una clase es una abstracción; un objeto es una manifestación concreta de esa abstracción.
  - *Entre interfaz e implementación:* una interfaz declara un contrato; ua implementación representa una realización del mismo.n
  - *Entre tipo y rol:* el tipo declara la clase de una entidad (como un objeto, un atributo o un parámetro); un rol

describe el significado de una entidad en un contexto, como una clase, un componente o una colaboración.

- ***Mecanismos de extensibilidad:*** permiten configurar y extender UML para las necesidades de un proyecto. Y adaptarlo a nuevas tecnologías de SW.
  - *Estereotipo:* extiende el vocabulario de UML, permitiendo crear nuevos bloques de construcción que derivan de los existentes, pero son específicos a un problema.
  - *Valor etiquetado:* extiende las propiedades de un estereotipo de UML, permitiendo añadir nueva información en la especificación de ese estereotipo.
  - *Restricción:* extiende la semántica de un bloque de construcción de UML, permitiendo añadir nuevas reglas o modificar las existentes.

PROCESO UNIFICADO DE DESARROLLO DE SOFTWARE

Proceso unificado de desarrollo: Es un conjunto de actividades o procesos o tareas que logran transformar los requerimientos de un cliente en un sistema de SW.

El **proceso unificado** (PUD): Es un marco de desarrollo de software. Se define como un entorno genérico de proceso que se puede especializar según las necesidades que tengamos en cada proyecto. Es un proceso de desarrollo de software que utiliza UML para preparar todos los esquemas de un sistema de software.

Características del PUD:

- *Dirigido por CU:* se utilizan para establecer el comportamiento deseado del sistema, para verificar y validar la arquitectura del sistema, para las pruebas y para la comunicación entre las personas involucradas en el proyecto. Los CU *guían el proceso de desarrollo*. Los CU se utilizan para capturar los requerimientos para desarrollarlos a través de los distintos WF en las distintas fases.
  El CU que mapea un requerimiento, lo llevamos como hilo conductor a lo largo de todos los WF del proceso, manteniendo una trazabilidad (responde al cliente sobre cierto requerimiento en qué etapa se encuentra, en que se transformó, nivel de avance)
- *Centrado en la arquitectura:* se utiliza para conceptualizar, construir, gestionar y hacer evolucionar el sistema en desarrollo. Nos da una visión general de que puede y no puede hacer el sistema, que es utilizado como cimiento. Se avanza con vistas arquitectónicas desde el WF de Requerimientos. Vamos haciendo evolucionar el sw teniendo en cuenta a la arq desde el 1er momento.
- *Iterativo:* involucra la gestión de un flujo de versiones ejecutables. Repite el proceso de trabajo
- *Incremental:* se integran las versiones, y cada nuevo ejecutable incorpora mejoras incrementales sobre los otros. Evolución del producto

Iterativo e incremental, ¿Que es? *Es el ciclo de vida* que el proceso unificado de desarrollo (PUD) *recomienda*. Tiene fases, las cuales identifican los momentos de evolución del proyecto, se trabaja con iteraciones, que entregan una versión del producto. Tenemos la libertad de elegir otro ciclo de vida, un proceso se puede trabajar con diferentes ciclos de vida.

Dirigido por casos de uso y centrado en la arquitectura van de la mano, xq la funcionalidad y la arquitectura se condicionan entre sí, a veces un RNF determina que la arquitectura se comporte de una determinada manera u otra. Ej: q el sistema sea web o Mobile, cliente liviano (no se debe instalar nada), es un RNF que impacta en la arquitectura.

Otra característica del PUD, es que cada WF genera como salida un modelo. El WF de diseño nos entrega 2 modelos, uno que se centra en el software (modelo de diseño) y otro en el hardware (modelo de despliegue), que hace falta para darle

soporte al sw que diseñamos. Nosotros no diseñamos hw, se encarga de la distribución de componentes de sw en que elementos de hw lo vamos a alojar.

Fases: intervalo de tiempo entre dos hitos importantes del proceso, muestran/determinan momentos de evolución del producto en el proyecto.

1. *Inicio:* contribuye a la planificación de los incrementos. El momento de evolución para salir del inicio y entrar a la fase de elaboración es cuando tiene claro el dominio del problema.
2. *Elaboración:* se especifican en detalle la mayoría de los CU (requerimientos) y se diseña la arquitectura. Contribuye a obtener una estructura robusta y estable, facilita una comprensión más profunda de los requerimientos
3. *Construcción:* se crea el producto.
4. *Transición:* el producto se convierte en versión beta. Un número reducido de usuarios con experiencia lo prueba e informa los defectos y deficiencias. Los desarrolladores los corrigen e incorporan mejoras.

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

INTERACCIONES

Interacción: comportamiento que incluye mensajes que se intercambian objetos en un contexto para lograr un propósito. Se utilizan para modelar los aspectos dinámicos de las colaboraciones en un sistema. Cada interacción puede modelarse de dos formas: destacando la ordenación temporal de los mensajes, o destacando la secuencia de mensajes en el contexto de una organización estructural de objetos. Una interacción establece el escenario para su comportamiento presentando todos los objetos que colaboran para realizar alguna acción.

- *contexto:* una interacción puede aparecer siempre que haya objetos enlazados entre sí en el contexto de un sistema o subsistema, en el contexto de una operación y en el contexto de una clase.
- *objetos:* son elementos concretos (del mundo real) o elementos prototípicos (instancias). Podemos encontrar instancias de clases, componentes, nodos y CU. Como las clases abstractas e interfaces no pueden tener instancias directas, podemos representarlas en una interacción (clases hijas concretas o que realicen la interfaz).
- *enlace:* conexión semántica entre objetos. Es una instancia de una asociación. Especifica un camino a lo largo del cual un objeto puede enviar un mensaje a otro objeto (o a sí mismo).
- *mensaje:* es la especificación de una comunicación entre objetos que transmite información, esperando que desencadene una actividad.

    acción: cuando se pasa un mensaje, su recepción produce una acción que puede causar un cambio en el estado del destinatario y en los objetos accesibles desde él.

    - llamada: invoca una operación sobre un objeto. Es el tipo más común de mensaje.
    - retorno: devuelve un valor al invocador.
    - envío: de una señal, que es un valor objeto que se comunica de manera asíncrona a un objeto destinatario. Después de enviarla, el que la envió continúa su propia ejecución. Cuando el destinatario la recibe, decide qué hacer con ella. Normalmente las señales disparan transiciones en la máquina de estados del receptor.
    - creación: crea un objeto.

- destrucción: destruye un objeto.

- *secuenciación:* flujo de mensaje que se forma cuando un objeto envía un mensaje a otro, y este a otro y así sucesivamente. El inicio de cada secuencia se origina en algún proceso o hilo que la contiene. Los mensajes se ordenan en secuencia conforme sucedan en el tiempo.
- *creación, modificación y destrucción:* los objetos y enlaces duran todo el tiempo que dura la interacción, algunas interacciones puede conllevar la creación y destrucción de ellos. En un diagrama de secuencia se representa explícitamente a través de sus líneas de vida y en un diagrama de comunicación se indica con notas.
- *representación:* los diagramas de secuencia y de comunicación son similares, podemos partir de uno y transformarlo en el otro, aunque a veces muestran información diferente.

MÁQUINAS DE ESTADOS

Es un comportamiento que especifica la secuencia de estados por las que pasa un objeto durante su vida, en respuesta a eventos, junto con sus respuestas a esos eventos. Se utilizan para modelar los aspectos dinámicos de un sistema, lo que implica modelar la vida de las instancias de una clase, un CU o un sistema completo, las cuales pueden responder a eventos tales como señales, operaciones o el paso del tiempo. Mientras una *interacción* modela una sociedad de objetos que colaboran para llevar a cabo una acción, una *máquina de estados* modela la vida de un único objeto, sea una instancia de una clase, un CU o un sistema completo. Durante su vida, un objeto está expuesto a varios tipos de eventos y como respuesta, reacciona con alguna acción, y *cambia su estado* a un nuevo valor.

Efecto: es la especificación de la ejecución de un comportamiento en una máquina de estados. Conllevan la ejecución de acciones que cambian el estado de un objeto o devuelven valores.

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

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

Transición: es una relación entre 2 estados que indica que un objeto que esté en el 1ero realizará ciertas acciones y entrará en el 2do cuando ocurra un evento especificado y se satisfagan ciertas condiciones especificadas.

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

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

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

Estado: un objeto permanece en un estado durante una cantidad de tiempo finita. Partes:

- *nombre:* cadena de texto que distingue al estado de otros.
- *efectos de entrada/salida:* acciones ejecutadas al entrar y salir del estado.
- *transiciones internas:* se manejan sin causar un cambio de estado.

- *subestados:* estructura anidada de un estado que engloba subestados disjuntos o concurrentes.
- *eventos diferidos:* se posponen y se añaden a una cola para ser manejado por el objeto en otro estado.
  - *estado inicial:* indica el punto de comienzo por defecto para la máquina.
  - *estado final:* indica que la ejecución de la máquina o del estado que lo contiene finalizó.

Transiciones: se representa con una línea continua desde el estado origen hacia el destino. Una auto transición es una transición donde el estado origen y el estado destino son el miso. Tiene cinco partes:

- *estado origen:* el afectado por la transición.
- *evento de disparo:* provoca la transición.
- *condición de guarda:* expresión booleana que se evalúa cuando la transición se activa por el evento de disparo.
- *efecto:* comportamiento ejecutable, como una acción.
- *estado destino:* estado activo tras completarse la transición.

WF DE ANÁLISIS en el PUD

Se analizan los requerimientos capturados en el WF de requerimientos, refinándolos y estructurándolos, con el *objetivo* de conseguir una comprensión de los mismos de una forma más precisa y una descripción fácil de mantener y que nos ayude a estructurar el sistema entero, analizándolos mayor profundidad. Habla en términos conceptuales (lógicos).

Workflows del PUD

- WF de Modelado de negocios (Opcional)
- WF de Requerimientos → Modelan el dominio del problema
- WF de Análisis (Puede ser Opcional) → Modela el dominio de la solución
- WF de Diseño → Modela el dominio de la solución
- WF de Implementación
- WF de Prueba → 4 niveles de testing (Prueba unitaria, integración, de sistema, aceptación de usuario)
- WF de Despliegue (Opcional) → Transferir la versión del producto al usuario final

Beneficio de porque hacer análisis:

★ Podemos analizar gran parte del sistema sin grandes costes. Se puede utilizar el resultado para planificar el trabajo de diseño e implementación subsiguiente en varios incrementos sucesivos, donde cada uno diseña e implementa una pequeña parte del sistema.

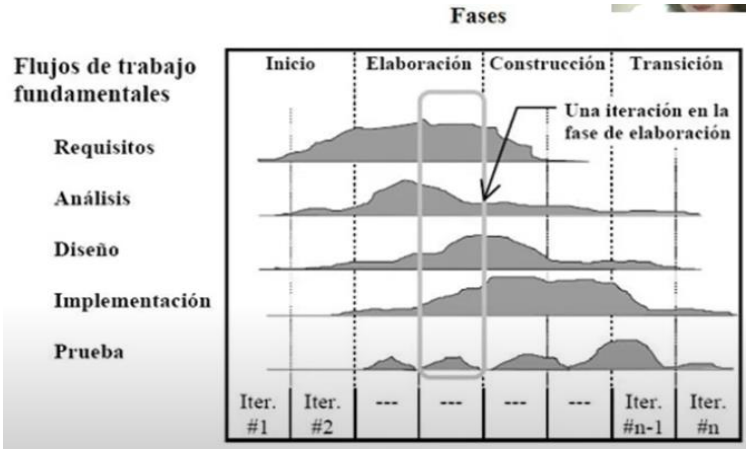
★ El análisis proporciona una visión más general del sistema, ya que al no hacer análisis puede ser difícil de obtener mediante el estudio de los resultados del diseño y la implementación, ya que son muy detallados. Una visión general puede ser muy valiosa para recién llegados al sistema o para desarrolladores que en general mantienen el sistema.

★ El modelo de análisis puede proporcionar una vista conceptual, precisa y unificadora de las implementaciones alternativas.

★ El sistema se construye utilizando un sistema heredado complejo, cuya reingeniería puede hacerse en términos del modelo de análisis donde los desarrolladores puedan comprender el sistema heredado sin profundizar en detalles de diseño e implementación y utilizarlo como un bloque de construcción reutilizable.

¿Cuál es el rol del análisis en el ciclo de vida iterativo e incremental del PUD?

El foco del análisis está puesto en las primeras iteraciones de la fase de **elaboración** (donde tiene el pico máximo de la curva de análisis), tiene incidencia en la fase de inicio, pero conforme va saliendo de la fase de elaboración ya no tiene tanta incidencia en el de construcción. Esta curva del análisis va cambiando de acuerdo a las 3 opciones que tenemos de uso del análisis en el contexto de Proceso Unificado



Cómo utilizar el análisis: tres formas en que el proyecto puede utilizar el modelo de análisis:

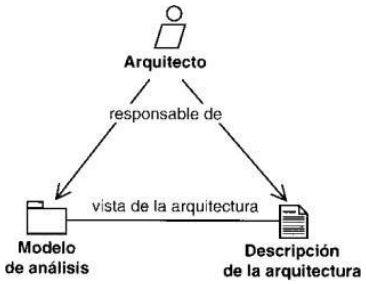
1. **Hacer y mantener**: Para describir los resultados de análisis, manteniendo este modelo a lo largo de todo el ciclo de vida del SW.
2. **Hacerlo y no mantenerlo**: Para describir los resultados del análisis, considerando el modelo como una herramienta intermedia o transitoria, al cerrar la idea de arquitectura y diseño, ya no mantenemos más el modelo de análisis. Cuando el diseño e implementación están en marcha durante la fase de construcción, se deja de actualizar el análisis. Suele ser la más usada, ya que el análisis se centra en la fase de elaboración.
3. **No hacer**: Significa fusionar el análisis con el diseño. Es decir que vamos a resolver los RF, RNF y restricciones de negocio en un solo modelo. Al no tener separado lo funcional lo hace más complejo de comprender.
   Es la opción por la cual opta el RUP.
   Al tenerlo separado puedo elegir para mi modelo de análisis diferentes modelos de diseño al cual implementarlo.
   Si quiero migrar a otra tecnología utilizando estos modelos, voy a tener que separar el análisis del diseño para operar con lo que necesitamos. Nunca hay un modelo de análisis, *hay un modelo de diseño con análisis embebido.*

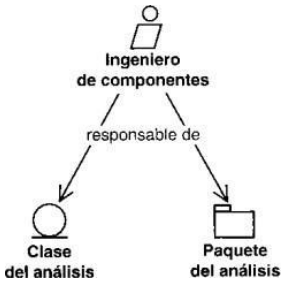
Trabajadores

- *Arquitecto:* Comienza con la creación del modelo y responsable de la integridad del mismo, garantizando que sea correcto, consistente y legible como un todo (realiza la funcionalidad descrita en el modelo de CU, y sólo esa). También es responsable de la arquitectura, de la existencia de sus partes SPA tal como se muestran en la vista de la arquitectura del modelo. No es responsable del desarrollo y mantenimiento continuo de los artefactos.
  Tiene responsabilidad por el planteo general del modelo de análisis, pero no es el responsable directo de la generación de los artefactos principales del análisis que son las realizaciones de caso de uso y las clases, es el responsable de delinearlas (identifica las clases de entidad), pero sí es responsable de la **vista arquitectónica del análisis**, ya que es un proceso centrado en la arquitectura (una de las 3 características).
- *Ingeniero de CU:* Es responsable de la realización de caso de uso de análisis en término de las clases de análisis participantes, garantizando que cumplen los requerimientos. Modela los escenarios más importantes, no modela todos, identifica clases de control e interfaz. Toma como entrada el modelo de casos de uso, requisitos, modelo de dominio y la vista del modelo de análisis. Tiene las 2 vistas, estática (diagrama de clases) y dinámica (diagrama de comunicación y secuencia)

Autor: Andermatten – Martinez – Sánchez Díaz

❖ ***Ingeniero de componentes:*** Especifican los requerimientos y los integran en cada clase creando responsabilidades, atributos y relaciones *consistentes* para cada clase, según las realizaciones de CU en las que participa. Se encarga de mantener la integridad de los paquetes de análisis y clases de análisis.





Artefactos de análisis:

❖ ***Modelo de análisis (Artefacto más Importante):*** Es un modelo de objetos lógico o conceptual de la solución del sistema(Se resuelve aspectos funcionales) que ayuda a refinar y estructurar los requerimientos (RF), pospone los RNF y restricciones. También proporciona una estructura centrada en el mantenimiento, la flexibilidad ante los cambios y la reutilización, la cual sirve como entrada para el diseño e implementación, ya que se intenta preservarla. Por esto se considera una primera aproximación al diseño.

Entrada => Modelo de Dominio (Vista de clases) y Descripción de Casos de Uso
Salida => Modelo de Análisis ->

El modelo de análisis hace abstracciones evitando algunos problemas y requerimientos que es mejor posponer al diseño y a la implementación. Por ello, no siempre se puede conservar la estructura proporcionada, ya que el diseño debe considerar la plataforma de implementación.

MODELO DE CASOS DE USO	MODELO DE ANÁLISIS
• descrito en lenguaje del cliente: intuitivo pero impreciso.	• descrito en lenguaje del desarrollador.
• vista externa del sistema.	• vista interna del sistema.
• estructurado por Casos de Uso.	• estructurado por clases y paquetes estereotipados.
• utilizado como contrato entre el cliente y los desarrolladores sobre que debería y que no debería hacer el sistema.	• utilizado por los desarrolladores para comprender cómo debería darse forma al sistema, cómo debería ser diseñado e implementado.
• puede contener redundancias, inconsistencias entre requerimientos.	• no debería contener redundancias ni inconsistencias entre requerimientos.
• captura la funcionalidad del sistema, RF y RNF, incluida la SPA (significativo para la arquitectura)	• esboza cómo llevar a cabo la funcionalidad en el sistema, incluida la SPA; sirve como una primera aproximación al diseño.
• define CU que se analizarán más profundamente en el análisis.	• define realizaciones de CU, donde cada una representa el análisis de un CU del modelo de CU.

❖ ***Descripción de la arquitectura (vista del modelo de análisis):*** Contiene una vista de la arquitectura del modelo de análisis, que muestra sus artefactos SPA (paquetes, clases, realizaciones de CU).

- ***Realización de CU-análisis:*** Es una colaboración dentro del modelo de análisis que describe cómo se lleva a cabo y se ejecuta un CU en término de clases y objetos del análisis en interacción. Poseen una descripción textual del flujo de sucesos, **diagramas de clases** que muestran sus clases de análisis participantes, y **diagramas de interacción** que muestran la realización de un escenario particular del CU en términos de interacciones de objetos del análisis. Se centra en los RF, posponiendo los RNF para el diseño y la implementación.

- ***Clase de análisis:*** representa una abstracción de clases y/o subsistemas del diseño del sistema. Se centra en el tratamiento de los RF y pospone los RNF. Son evidentes en el contexto del dominio del problema.
  - Sus ***métodos*** se definen mediante responsabilidades en un alto nivel y menos formal.
  - Define ***atributos*** en un alto nivel: los tipos de los atributos son conceptuales y reconocibles en el dominio del problema. En el diseño los tipos pertenecen al lenguaje de programación (string, char. Ej Nombre:string).
  - Participa en ***relaciones***, que son más conceptuales que las de diseño e implementación.

Tres estereotipos estandarizados en UML que ayudan a los desarrolladores a distinguir el ámbito de las clases:

- ***Interfaz:*** Modelan la interacción entre el sistema y sus actores (humano, sistema, tiempo, dispositivo), que implica recibir y enviar información y peticiones entre actores y sistemas externos. Representan abstracciones de ventanas, formularios, paneles, interfaces, etc. Deberían asociarse con al menos un actor y viceversa.
- ***Entidad:*** Modelan información y el comportamiento asociado a algún concepto que posee una vida larga y es persistente, como una persona, un objeto o suceso del mundo real. Usualmente se derivan de las clases de dominio. Modelan información relevante para el dominio y que dura en el tiempo, siendo candidatas a ser persistentes mediante bases de datos.
- ***Control (Gestor):*** Es el que maneja y coordina el control de objetos y se usan para encapsular el control de un CU concreto. También para representar derivaciones y cálculos complejos, como la lógica de negocio. Modelan los aspectos dinámicos del sistema, ya que manejan y coordinan las acciones y flujos de control principales, y **delegan trabajo a otros objetos**. Son el nexo entre las clases de entidad y de interfaz, ya que éstas no se relacionan directamente.

- ***Paquetes del análisis:*** Proporcionan un medio para agrupar y organizar los artefactos del modelo en piezas manejables. Puede estar compuesto de clases, realizaciones de CU y de otros paquetes (recursivamente). Deberían ser **cohesivos** (contenidos fuertemente relacionados, deben modelar el mismo concepto) y **débilmente acoplados** (mínima dependencia). Tienen las siguientes características:
  - Pueden representar una separación de intereses de análisis.
  - Deberían crearse basándose en los RF y en el dominio del problema.
  - Probablemente se convertirán en subsistemas o se distribuirán entre ellos.

Actividades (flujo de trabajo)

1. ***Análisis de la arquitectura:*** Su propósito es esbozar el modelo de análisis y la arquitectura mediante la identificación de paquetes de análisis, clases de análisis evidentes y requerimientos especiales comunes.

❖ *Arquitecto:*

- ***Identificación de paquetes del análisis*** *F*orma de dividir el trabajo. Una identificación inicial de los paquetes se basa en los RF y el dominio del problema: se asigna un número de CU a un paquete y luego se realiza la funcionalidad correspondiente al mismo. Pueden hacerse para:
  - Dar soporte a un determinado proceso de negocio
  - Dar soporte a un determinado actor del sistema
- ***Identificación de clases de entidad evidentes*** basado en las clases del dominio o entidades de negocio que se identificaron en la captura de requerimientos.
- ***Identificación de requerimientos especiales comunes (RNF)*** se pospone para el diseño e implementación.

2. ***Analizar un CU:***

❖ *Ingeniero de CU:*

- ✔ ***Identificación de clases de análisis****, clases de entidad, control e interfaz,* cuyos objetos son necesarios para realizar el CU. Se esbozan sus nombres, responsabilidades, atributos y relaciones.
- ✔ ***Descripción de interacciones entre objetos del análisis****:* se distribuye el comportamiento del CU entre los objetos del análisis que interactúan, mediante diagramas de colaboración que contienen instancias de actores participantes, objetos del análisis y sus enlaces.
- ✔ ***Captura de requerimientos especiales****:* Se los captura, pero se postergan para el diseño e implementación.

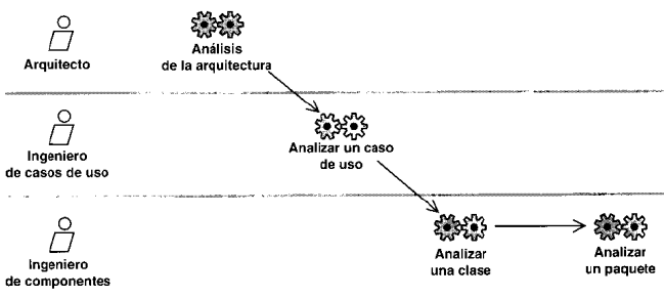
3. ***Analizar una clase:***

❖ *Ingeniero de componentes*

Identificar responsabilidades de una clase de análisis y mantenerlas, combinando todos los roles que cumple en diferentes realizaciones de CU.

- ✔ ***Identificar atributos*** que especifican propiedades de una clase de análisis, y son necesarios para las responsabilidades de su clase.
- ✔ ***Identificar asociaciones, agregaciones y generalizaciones,*** se estudian los diagramas de colaboración para determinar las asociaciones necesarias.
- ✔ ***Captura de requerimientos especiales****:* se postergan para el diseño e implementación.

4. ***Analizar un paquete:*** Se encarga de mantener su integridad, describiendo las dependencias de forma que pueda estimarse el efecto de los cambios futuros y garantizando: que sea tan independiente de otros paquetes como sea posible y que cumple su objetivo de realizar algunas clases de dominio o CU.



PATRONES GRASP

Son patrones que nos ayudan a modelar las realizaciones de caso de uso de análisis, para ayudarnos a definir a donde vamos a determinar la responsabilidad, o sea donde vamos a asignar un método en que clase es más conveniente ubicar un determinado comportamiento. (meles)

Responsabilidad: "un contrato u obligación de un clasificador". Están relacionadas con las obligaciones de un objeto en cuanto a su comportamiento. **No** es lo mismo que un ***método***: estos se implementan para llevar a cabo responsabilidades. Las responsabilidades se implementan utilizando métodos que actúan solos o colaboran.

❖ *del conocer:*

- datos privados propios encapsulados: a través de los métodos accedemos a los atributos (nombre, apellido, DNI).
- objetos relacionados: Objetos con los que está relacionado. *Son los métodos que me permiten a mi tener visibilidad de los objetos con los que yo ya tengo una relación*. Permite conocer los atributos de las clases relacionadas (ej get())
- Cosas que puede derivar o calcular: Los atributos que son derivables o los que se pueden calcular, no deben existir como atributos en una clase, tiene que estar el método que calcule ese atributo. Ej FechaNacimiento(), no usar edad como atributo.

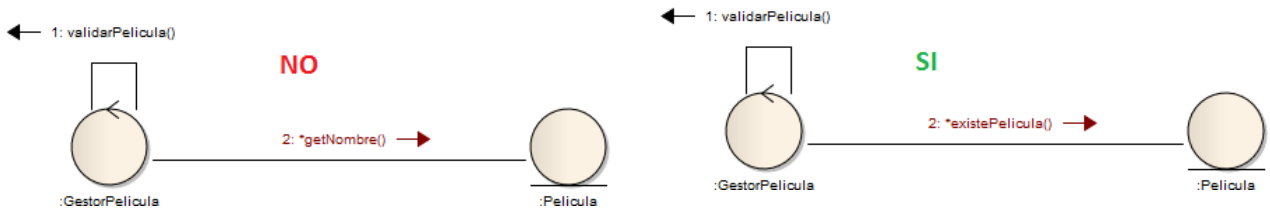
❖ *del hacer:*

- Hacer algo él mismo: crear un objeto o hacer un cálculo, responder peticiones.
- Iniciar una acción en otros objetos: delegar responsabilidades. Ej: Cuando un gestor crea un objeto
- Controlar y coordinar actividades en otros objetos: Clases de control, Gestores

Patrón: es un par problema/solución con nombre que se puede aplicar en nuevos contextos, con consejos acerca de cómo aplicarlo en nuevas situaciones y discusiones sobre sus compromisos. Lo importante de los patrones no es expresar nuevas ideas de diseño, sino lo contrario: pretenden codificar conocimientos, estilos y principios existentes y que se han probado que son válidos. Todos los patrones, idealmente, tienen nombres sugerentes que apoyan la identificación e incorporación de ese concepto en nuestro conocimiento y memoria, y facilita la comunicación.

Patrones GRASP (Patrones Generales de Asignación de Responsabilidades en el SW): describen los principios fundamentales del diseño de objetos y la *asignación de responsabilidades en las clases* (conocer y hacer), expresados como patrones (5).

❖ *Experto en información:* asignar una responsabilidad a la clase que tiene la información necesaria para realizarla. Poner el comportamiento (métodos) lo más cerca posible de los datos, en la clase que los tiene.

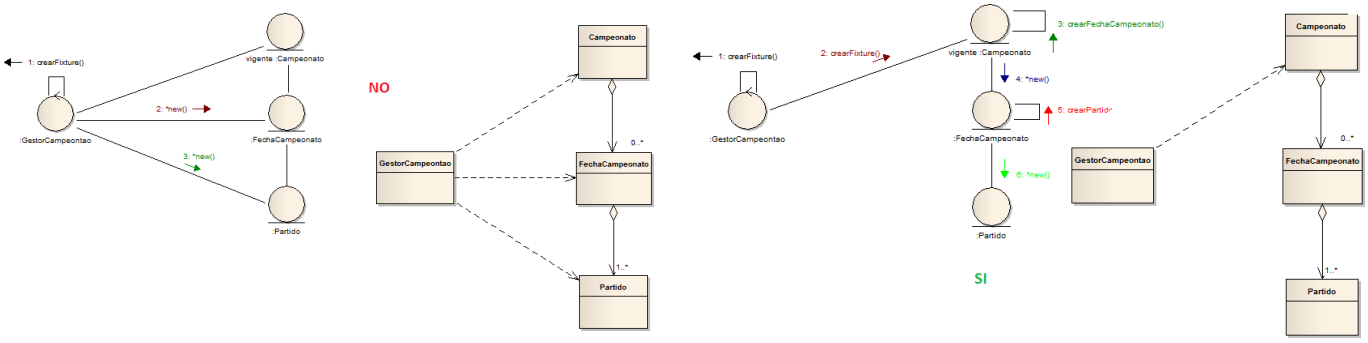


Beneficios:

- mantiene el encapsulamiento de la información: los objetos utilizan su propia información para realizar las tareas. Esto conlleva a un *bajo acoplamiento*, dando lugar a sistemas más robustos y fáciles de mantener.
- se distribuye el comportamiento entre las clases que contienen la información requerida: se estimula las definiciones de *clases más cohesivas* y ligeras que son más fáciles de entender y mantener.

❖ *Creador:* Es un tipo especial del patrón experto. Resuelve a quien le da la responsabilidad de invocar ese new(), cada clase tiene su new(). Una clase instancia un objeto a través del new(). La decisión es quien invoca el constructor de esa clase para crear el objeto, esto nos ayuda a decidir el patrón creador. Ej. CrearPelicula() como

self del Gestor Pelicular, tiene la lógica y la invocación al new() de la clase Película, entonces yo decido por patrón creador que le voy a asignar la responsabilidad de creación de objeto película al gestor película. Se priorizan las relaciones de dominio



Asignar a la Clase A la responsabilidad de crear una instancia de la clase B:

- *A agrega los objetos de B: A es el todo, esta formado por objetos de B. En el caso de agregación, la responsabilidad de las partes de B, se las tengo q dar a A (al todo). Tiene responsabilidad de creación sobre sus partes.*
- *A contiene los objetos de B: Relación de composición, contención. El todo tiene responsabilidad de creación de sus partes*
- *A tiene los datos de inicialización de B: Son los parámetros que yo le paso en el new() al objeto B cuando lo voy a crear. Así A es un experto respecto de la creación de B. Lo tiene q crear xq tiene la información q hace falta para crearlo, en este caso serían los valores de los atributos que le van a pasar como parámetro en el constructor. Si el gestor crea todos los objetos se crea un problema de acoplamiento*

Beneficios: se eliminan dependencias innecesarias, soportando *bajo acoplamiento* y mayores oportunidades para *reutilizar*.

❖ *Bajo acoplamiento:* asignar una responsabilidad de manera que el acoplamiento permanezca bajo. *La idea es tener las clases lo menos ligadas entre sí como sea posible, para que en caso de producirse una modificación en alguna se tenga la mínima repercusión posible en el resto, potenciando la reutilización, y disminuyendo la dependencia entre las clases.* El caso extremo de Bajo Acoplamiento es cuando no existe: esto no es deseable porque una metáfora central de la tecnología de objetos es un sistema de objetos conectados que se comunican mediante el paso de mensajes. Un grado moderado de acoplamiento es normal y necesario si quiere crearse un sistema OO, donde los objetos colaboran entre sí:

Acoplamiento: es una medida de la fuerza con que un elemento está conectado a otros elementos. Nivel de relación entre las clases.

* Un elemento con bajo acoplamiento no depende de demasiados otros.
* Una clase con alto acoplamiento confía en muchas otras. Tales clases podrían tener problemas:

χ los cambios en las clases relacionadas fuerzan cambios locales.
χ son difíciles de entender de manera aislada.
χ son difíciles de reutilizar ya que su uso requiere la presencia adicional de las clases de las que depende.

Beneficios:

- no afectan los cambios en otros componentes.
- fácil de entender de manera aislada.
- conveniente para reutilizar.

❖ *Alta cohesión:* asignar una responsabilidad de forma que la cohesión permanezca alta. *Cada atributo y método tiene que tener sentido en el concepto de la clase. Una alta cohesión funcional provoca mayor cantidad de clases.* Un concepto, una clase. Cada clase debe modelar un concepto

Cohesión: es una medida de la fuerza con la que se relacionan los objetos y de grado de focalización de las responsabilidades de un elemento. Un elemento con responsabilidades altamente relacionadas, y que no hace una gran cantidad de trabajo, tiene *alta cohesión*. Una clase con *baja cohesión* hace muchas cosas no relacionadas, o hace demasiado trabajo, no son convenientes y tienen problemas:

- χ son difíciles de entender, reutilizar y mantener.
- χ son delicadas, constantemente afectadas por los cambios.

Beneficios:

- se incrementa la claridad y facilita la compresión del diseño.
- se simplifican el mantenimiento y las mejoras.
- se soporta a menudo bajo acoplamiento.
- incrementa la reutilización porque una clase cohesiva se puede utilizar para un propósito muy específico.

❖ *Controlador:* asignar la responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que:

- representa el sistema global, dispositivo o subsistema.
- representa un escenario de CU en el que tiene lugar el evento del sistema.

Es un patrón que sirve como intermediario entre una interfaz y el algoritmo que la implementa (entidades), el controlador recibe la solicitud del servicio desde la capa de UI y coordina su realización delegando a otros objetos. *Sugiere que la lógica de negocios debe estar separada de la capa de presentación* para aumentar la reutilización de código y tener un mayor control. Los objetos interfaz y la capa de presentación no deberían ser responsables de llevar a cabo los eventos del sistema. Las operaciones del sistema se deberían manejar en la lógica de la aplicación.

Controlador: es un objeto que no pertenece a la interfaz de usuario, responsable de recibir o manejar un evento del sistema. Un controlador define el método para la operación del sistema. Debería delegar en otros objetos el trabajo que necesita hacer, coordina o controla la actividad. Hay dos categorías de controlador:

- χ *de fachada:* representa al sistema global, dispositivo o subsistema. Se usan cuando no existen "demasiados" eventos, o la UI no puede redireccionar mensajes de los eventos a controladores alternativos.
- χ *de CU:* hay un controlador diferente para cada CU. No es un objeto de dominio, es una construcción artificial para dar soporta al sistema (*clase de fabricación pura:* creación arbitraria del diseñador).

Beneficios:

- aumenta el potencial para reutilizar y las interfaces conectables: delegar la responsabilidad de una operación del sistema a un controlador ayuda a la reutilización de la lógica en futuras aplicaciones, ya que la lógica no está ligada a la interfaz, puede sustituirse por otra interfaz.
- razonamiento sobre el estado de los CU: es necesario asegurar que las operaciones del sistema están en una secuencia válida, o son capaces de razonar sobre el estado actual de la actividad y operaciones del CU en marcha

Unidad Nro. 2: Diseño de Sistemas de Información Orientado a Objetos con UML

WORKFLOW DE DISEÑO

¿Qué es diseñar?: Es un proceso iterativo de transformar un modelo lógico en un modelo físico (componentes de código, lenguaje de programación, tecnologías, seguridad, tiempos de rta), modelamos el sistema y definimos su forma (incluida la arquitectura) para que soporte todos los requerimientos, incluyendo los RNF y otras restricciones, entrando en el dominio de la solución. Generando un modelo de diseño, más detallado que nos acerca a la implementación. Tiene suficiente detalle que nos va a permitir su realización física.

En el WF de Diseño, se toma el modelo de análisis como entrada obtenido en el WF de Análisis y se aplican los RNF obtenidos en el WF de Requerimientos, buscando generar un modelo que respete la estructura del análisis tanto como sea posible al momento de darle forma al sistema. Es el proceso de decidir cómo se va a construir el producto final.

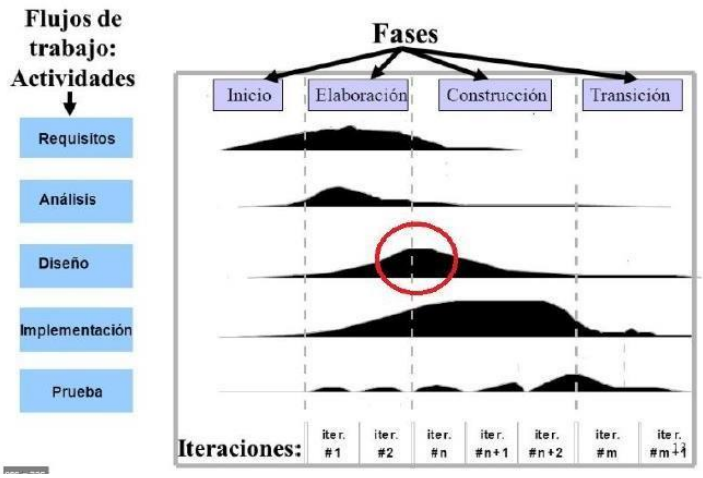
El diseño es el WF que se encarga de avanzar en el refinamiento de la solución que empezó a estructurarse en el WF de análisis.

Aplicamos Patrones de diseño con el Objetivo de rediseñar y mejorar la estructura que se había conseguido en el WF de análisis, hay muchos aspectos de la solución que no había sido tenido en cuenta. Nivel de cohesión y acoplamiento lo más óptimo posible

Propósito del diseño: ¿Para qué diseñamos?

- ➢ Comprender en profundidad los aspectos relacionados a los RNF y restricciones tecnológica(lenguaje, componentes reutilizables, SOP, tecnologías de distribución, concurrencia, interfaz de usuario).
- ➢ Refinar los requerimientos para subsistemas, clases e interfaces.
- ➢ Crear una entrada para el WF de implementación
- ➢ Descomponer el trabajo de implementación en partes más manejables por diferentes equipos de trabajo.
- ➢ Identificar las interfaces entre los subsistemas (que pantalla va para cada subsistema, diagrama de componentes).
- ➢ Crear una abstracción de la implementación del sistema. diagrama de despliegue (plano)

Rol en el PUD: El pico más fuerte está en las últimas iteraciones de la fase de elaboración (contribuye a una arquitectura estable y robusta) y las primeras de la construcción (crea un plano para la implementación). Una vez que los requerimientos son bien entendidos y la arquitectura está bien definida, el centro de atención se desplaza a la implementación.



Autor: Andermatten – Martinez – Sánchez Díaz

Aspectos del SW que se diseñan

- *Arquitectura:* Es el conjunto de decisiones significativas respecto a resolver requerimientos de calidad (RNF) y funcionalidad del producto de SW.
  Es diseño general, estratégico (No hay detalle) hay definiciones de cómo resolver requerimientos no funcionales frente a un contexto de infraestructura específico. Define la relación entre los principales elementos estructurales del programa. Toma los RNF SPA y los aplica al modelo de análisis. Se representa a través de Vistas (diagramas UML), 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 del Análisis, se obtienen las Realizaciones de Casos de Uso de Diseño luego de considerar y aplicarles los RNF.
- *Datos o Diseño de Persistencia:* Implica analizar nuestras clases, definir cual de esas clases necesitan persistencia, en que almacenamiento persistente la 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 software.
- *Interfaz o IHM:* Es una disciplina que se encarga de evaluar todos los aspectos que tiene que ver entre las personas y el SW. Hace referencia a lo que el usuario ve (UI= interfaz de usuario) y su interacción (UX = experiencia de usuario). Se utilizan patrones para el diseño de interfaces.
- *Entrada/Salida:* describe cómo se ingresa información al SW y cómo se presentarán las salidas del mismo, en lotes (se acumulan transacciones para procesarlas después), en línea (la transacción se procesa en el momento que ocurre), en tiempo real (son sistema en línea, pero puede modificar el ambiente donde está inmerso, estricto en tiempo de rta).
- *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.

Entrada: Modelo de Análisis (WF de Análisis) + RNF (WF de Requerimientos)
Salidas: Modelo de Diseño + Modelo de Despliegue + Descripción de la Arquitectura (Vistas de Diseño y Despliegue)

Modelo de Diseño: es un modelo de objetos que describe la realización física de los CU, centrando la atención en cómo los RF y RNF, junto a otras restricciones de la 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.

Descripción de la Arquitectura:
- *Como vista de Diseño:* Contiene los artefactos relevantes para este modelo (subsistemas, sus interfaces y dependencias entre ellos). Constituye la estructura fundamental del sistema. Las clases de diseño fundamentales y las realizaciones de CU-Diseño que describen alguna funcionalidad crítica.

- *Como vista de Despliegue:* contiene los artefactos relevantes para la arquitectura, incluyendo la correspondencia de los artefactos sobre los nodos.

Trabajadores y artefactos

- **Arquitecto:** Es el responsable de la arquitectura, la vista arquitectónica y de la integridad de los 2 modelos de salida, de diseño y despliegue, garantizando que sean correctos, consistentes y legibles como un todo, no hace el trabajo detallado. Los modelos son correctos cuando realizan la funcionalidad, y sólo la funcionalidad, descrita en el modelo de CU, en los requerimientos adicionales y en el modelo de análisis. El modelo de despliegue se diseña en el WF de Diseño, y se ejecuta en el WF de Despliegue.
  - Modelo de diseño: Enfocado al software, tiene clases de diseño, componentes, interfaces, realizaciones de CU de diseño y la parte SPA de ese modelo de diseño que está representado por el artefacto llamado descripción de la arquitectura
  - Modelo de despliegue: Enfocado al hardware, contiene los recursos de hardware, comunicaciones e infraestructura que hacen falta para que el sw que diseñamos pueda funcionar cumpliendo con los requerimientos, teniendo en cuenta la parte SPA. No creamos hardware sino nos enfocamos en los elementos que son necesarios para que nuestro sistema funcione.
  - Descripción de la arquitectura (vista del modelo de diseño) Diagrama de componentes
  - Descripción de la arquitectura (vista del modelo de despliegue) Diagrama de despliegue (SW y HW)

- **Ingeniero de CU:** Responsable de la realización e integridad de los CU-diseño, garantizando que cumplen con los requerimientos que se esperan de ellos. Una realización de CU-Diseño debe realizar correctamente el comportamiento de su correspondiente realización de CU-Análisis y del CU del modelo CU, y sólo esos comportamientos.
  Toma las realizaciones de CU de análisis y las rediseña aplicando cambios teniendo en cuenta RFN y restricciones, construyendo los procesos de diseño (Realizaciones de CU de Diseño)
  - Realizaciones de CU-Diseño: Proporciona una realización física de la realización de CU de análisis.

- **Ingeniero de Componentes:** define y mantiene las operaciones, métodos, atributos, relaciones y requisitos de implementación de clases de diseño, garantizando que cumple lo que se espera de ellas en la Realización de CU. Puede mantener también la integridad de subsistemas de diseño, interfaces y componentes (porción de código).
  - Clases de diseño: Debe ser completa y suficiente, sencilla, alta cohesión y bajo acoplamiento
  - Subsistemas de diseño: Forma de organizar los artefactos en piezas más manejables
  - Interfaces: Especificar las operaciones que proporcionan las clases y los subsistemas de diseño

Comparación de con el WF de análisis:

Análisis	Diseño
Modelo Conceptual	Modelo Fisico
Genérico respecto al diseño	Específico para una implementación
Tres estereotipos	Cualquier número de estereotipos
Menos formal	Más formal
Menos dinámico	Más dinámico
Más barato de desarrollar	Más Caro de desarrollar
Puede no mantenerse	Debe mantenerse en todo el ciclo de vida
Define una estructura que es entrada esencial para modelar el sistema.	Da forma al sistema mientras que intenta preservar la estructura definida por el modelo de analisis

ESTRATEGIAS DE PROTOTIPADO

Estrategia de prototipado: Es la elección de este tipo de ciclo de vida que se recomienda elegir al implementar un proyecto complejo, con dominio no familiar, que utilizará una tecnología desconocida. Requiere el uso de prototipos en el diseño, la implementación y la validación de requerimientos.

Prototipo: Es un modelo o maqueta del sistema que se construye para comprender mejor el problema y sus posibles soluciones: evaluar mejor los requerimientos y probar opciones de diseño.
Consiste en una primera versión de un nuevo tipo de producto, que incorpora sólo algunas características del sistema final (incompleto)
Objetivo: aclarar los requerimientos de los usuarios y verificar la factibilidad del diseño del sistema.

Beneficios:

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

Desventajas/Riesgos/Críticas:

- χ Funcionalidad limitada.
- χ Poca fiabilidad.
- χ Características de operación pobres.
- χ Alto costo: 10% del presupuesto del proyecto
- χ El cliente cree que es el sistema funcional.
- χ Peligro de familiarización con malas elecciones iniciales.
- χ Difícil de administrar: se necesita mucha experiencia.

¿Cómo se utilizan? se presenta al cliente para que experimente, ayudándolo a establecer claramente los requisitos.

- En la ingeniería de requerimientos: ayuda a obtener y validar los requerimientos.
- En el diseño: ayuda a explorar soluciones de SW particulares y apoyar el diseño de interfaces de usuario.
- En el proceso de pruebas: para ejecutar pruebas back-to-back con el sistema que se entregará al cliente.
- Permiten a los usuarios ver como el sistema apoya a su trabajo: pueden adquirir nuevas ideas para los requerimientos, encontrar áreas fuertes y débiles en el SW, revelar errores y omisiones.
- Ayuda a los desarrolladores a: validar correcciones de la especificación, aprender sobre problemas que se presentarán en el diseño e implementación, mejorar el producto, examinar viabilidad y utilidad de la aplicación.

¿Cuándo se utilizan?

- el área de aplicación no está bien definida: dominio riesgoso, por dificultad o falta de tradición en su aplicación.
- el costo de rechazo si no se cumplen las expectativas del cliente es muy alto.
- es necesario evaluar previamente el impacto del sistema en los usuarios y la organización.
- se utilizan nuevos métodos, técnicas o tecnologías.
- no se conocen los requerimientos o es necesario evaluarlos.
- hay factores de riesgo alto asociados al proyecto.

Autor: Andermatten – Martinez – Sánchez Díaz

Clasificación:

❖ *Según su Propósito:*
- ***De interfaz de usuario****:* modelos de pantallas. Les permite a los usuarios ganar experiencia directa con la interfaz para juzgar su usabilidad.
- ***Funcional*** *(operacional)*: implementa funciones, comprueba si son apropiadas, se corrigen, refinan y agregan.
- ***Arquitectónicos****:* sirven para evaluar decisiones arquitectónicas de infraestructura, tecnológica e integración.
- ***Modelos de rendimiento****:* evalúan el rendimiento de una aplicación crítica (no sirven al análisis de requerimientos).

❖ *Según su Utilidad:*
- ***Rápidos/desechables****:* sirve al análisis y la validación de requerimientos, se redacta la ERS y se desecha. La aplicación se desarrolla con otro paradigma. *Problema:* cuando no se desecha y se convierte en el sistema final.
- ***Evolutivos****:* comienza con un sistema relativamente simple que implementa los requerimientos más importantes o mejor conocidos. Va cambiando en cuanto se descubren nuevos requerimientos y finalmente, se convierte en el sistema requerido. Se usa en el desarrollo Web y en aplicaciones de comercio electrónico.

❖ *Según su Alcance:*
- ***Vertical****:* desarrolla completamente alguna de las funciones. (desarrollo un CU completo)
- ***Horizontal****:* desarrolla parcialmente todas las funciones. (de varios CU solo hacer una parte)

Proceso de desarrollo de prototipos:

1. Establecer objetivos (*plan de construcción del prototipo*): debe ser claros desde el principio, ya que se puede malinterpretar la función del prototipo y no se obtengan los beneficios esperados.
2. Definir funcionalidad del prototipo (*definición general*): decidir que incluir y excluir, dejando de lado RNF y centrándose en la funcionalidad, reducir estándares, obviar manejo de errores, etc. dependiendo el tipo.
3. Desarrollar prototipo (*ejecutable*): construirlo.
4. Evaluar prototipo (*informe de evaluación*): analizar los resultados de su uso.

Técnicas de prototipado:

❖ *Prototipación conducida por guiones:* desarrollar un conjunto de guiones y pantallas utilizando una herramienta. Cuando el usuario interactúa, la pantalla cambia al siguiente display.
❖ *Programación visual:* utilizar un lenguaje diseñado para desarrollo rápido (ej: Python PyQT).
❖ *Prototipado basado en internet:* utilizar un navegador web y guiones asociados.

Estrategias para el desarrollo de prototipos

- *Prototipos para pantallas:* el elemento clave es el intercambio de información con el usuario.
- *Prototipos para procedimientos de procesamiento:* el prototipo incluye sólo procesos sin considerar errores.
- *Prototipos para funciones básicas:* solo se desarrolla el núcleo de la aplicación, es decir sólo los procesos básicos.

Tareas de los usuarios: identificar la finalidad del sistema, describir la salida, describir los requerimientos de datos, utilizar y evaluar el prototipo, identificar mejoras necesarias y documentar características no deseables.

Creencias erróneas sobre los prototipos

χ el desarrollo del prototipo es trivial
χ es sólo para aplicaciones pequeñas
χ es sólo para aplicaciones sencillas
χ la participación del usuario es simbólica

ESTRATEGIA DE ENSAMBLADO DE COMPONENTES

Estrategia de ensamblado de componentes: es una decisión arquitectónica y de diseño de la solución final del SW a construir que implica desde decidir implementar por componentes y definir la granularidad hasta su ensamblando final y prueba de integración, siendo un modelo evolutivo de desarrollo SW. Incorpora muchas características del modelo en espiral. Exige un enfoque iterativo para la creación del SW.

• comienza con la *identificación de clases candidatas*, examinando los datos que se van a manejar por parte de la aplicación y el algoritmo que se va a aplicar para conseguir el tratamiento, empaquetándolos en una clase.

• las clases (componentes) creadas en los proyectos de ingeniería del SW anteriores se almacenan en una *biblioteca de clases*. Una vez identificadas las candidatas, se examina la biblioteca para determinar si ya existen.

Ingeniería de SW basado en componentes: es un modelo evolutivo de desarrollo de SW basado en la reutilización de piezas de código. Es un paradigma de ensamblar componentes y escribir códigos para que estos funcionen. La complejidad de los sistemas computacionales actuales nos llevó a la reutilización de SW existente. El desarrollo de SW basado en componentes consiente en reutilizar piezas de código preelaborado (*componentes*) que encapsulan alguna funcionalidad expuesta a través de interfaces. La ISBC es el proceso de definir, implementar e integrar los componentes independientes en los sistemas. La ISBC se centra en el diseño y construcción de sistemas computacionales que utilizan componentes de SW reutilizables. Trabaja bajo la filosofía de "comprar, no construir". Proporciona ventajas significativas, ya que con la reutilización pueden llevar a una reducción del 70% del tiempo de desarrollo y más del 80% del costo del proyecto. La modularidad que ofrece permite evolucionar de manera *independiente* los componentes hacia una mejor calidad, simplificando las pruebas y el mantenimiento del sistema.

Fundamentos de la ISBC

- componentes independientes especificados por sus interfaces.
- estándares de componentes para facilitar la integración de componentes.
- middleware que brinda soporte para la integración de componentes.
- un proceso de desarrollo que empalma con la ISBC.

Principios de diseño que conforman la estrategia de ensamblado de componentes:

- Facilitar la reutilización: encapsular funcionalidad en componentes que se pueden utilizar en múltiples piezas de SW (***DRY***). Las plataformas de componentes ofrecen servicios estándar reduciendo costos de desarrollo.
- Independencia de componentes: sus ejecuciones no interfieren.
- Comunicación mediante interfaces bien definidas: se puede sustituir un componente por otro mientras ambos utilicen la interfaz adecuada. (***PTI***)
- Implementaciones ocultas de los componentes: pueden cambiar sin afectar el resto del sistema. (***EWV***)

Ventajas del ensamblaje de componentes:

- aumento de reutilización del SW.
- simplifica las pruebas: antes de probar el conjunto completo de componentes ensamblados.
- simplifica el mantenimiento del sistema: cuando hay un débil acoplamiento entre componentes, el desarrollador es libre de actualizar y/o agregar componentes según sea necesario, sin afectar otras partes del sistema.
- mayor calidad: un componente puede ser construido y mejorado continuamente.
- Proporciona bajo acoplamiento si se hace un buen uso

Problemas de la ISBC:

- integridad de componentes: ¿cómo puede ser confiable un componente sin su código fuente disponible?
- certificación de componente: ¿quién certificará su calidad?, no sabemos quien lo hizo ni de donde proviene
- predicción de propiedades emergentes de composiciones de componentes: ¿cómo se pueden predecir?
- requerimientos de compensaciones: ¿cómo se realiza el análisis de compensaciones entre componentes?

Autor: Andermatten – Martinez – Sánchez Díaz

Procesos de la ISBC: le dan soporte. Existen dos tipos de procesos:

- *Desarrollo para reuso:* componentes que se reutilizarán en otras aplicaciones, implica la generalización de los existentes. Mientras más general, mayor es la capacidad de reutilización. Es necesario un equilibrio entre la reutilización y usabilidad. Los componentes deberían:
  - reflejar abstracciones estables del dominio
  - esconder la representación del estado
  - ser lo más independientes posible
  - publicar excepciones en la interfaz del componente.

Cambios en los componentes para hacerlos reusables:

- eliminar métodos específicos de aplicación
- cambiar los nombres por más generales
- agregar métodos para brindar cobertura funcional
- hacer manejadores de excepción consistentes para todos los métodos
- agregar interfaz de configuración para permitir la adaptación de los componentes a diferentes situaciones
- integrar los componentes requeridos para aumentar la independencia.

Tipos de composición de componentes:

  - **Secuencial**: se ejecutan en secuencia. Se requiere códigos "pegamento" dado que no se llaman mutuamente.
  - **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".
  - **Aditiva**: 2 interfaces se unen para crear un componente nuevo. No son dependientes ni se llaman mutuamente.

- *Desarrollo con reuso:* se ocupa para desarrollar nuevas aplicaciones usando componentes y servicios existentes

Componentes: es una entidad ejecutable independiente que provee un servicio sin importar donde se está ejecutando o su lenguaje de programación. No necesita ser compilado y puede usarse con otros componentes. Se publica la interfaz del componente en términos de operaciones parametrizadas y su estado interno nunca se expone, por lo que todas las interacciones son a través de la interfaz publicada. Son como los "ingredientes de las aplicaciones", que se juntan y combinan para llevar a cabo una tarea. Son piezas de código preelaborado que encapsulan alguna funcionalidad expuesta a través de interfaces.

Características:

- *Estandarizado:* debe ajustarse a un estándar que define: interfaces, metadatos, documentación, composición e implementación.
- *Independiente:* debe ser factible componerlo e implementarlo sin usar otros componentes. Un componente funciona sin necesidad de otro
- *Componible:* todas las interacciones externas deben darse mediante interfaces definidas públicamente. Debe permitir acceso a información sobre sí mismo.
- *Implementable:* debe ser auto contenido, capaz de ejecutarse como entidad independiente.
- *Documentado:* debe implementarse en forma completa, especificando la sintaxis y semántica de todas sus interfaces. Así los usuarios deciden si el componente satisface sus necesidades.

Comprar componentes de terceros en lugar de desarrollarlos, posee algunas ventajas:

1. *Ciclos de desarrollo más cortos:* la adición de una pieza dada de funcionalidad toma menos tiempo.
2. *Mejor ROI:* el retorno sobre la inversión puede ser más favorable que desarrollando los componentes.
3. *Funcionalidad mejorada:* para usar un componente solo se necesita entender su naturaleza, no detalles internos. Que hace y no como lo hace.

PATRONES DE DISEÑO:

Patrón: Solución concreta a un problema bien identificado.

¿Qué es un patrón de diseño? Son genéricos, son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software.

Cada patrón describe un problema que ocurre una y otra vez en un determinado entorno, así como la solución a ese problema.

Nuestro diseño debe ser específico del problema que estamos manejando, pero lo suficientemente general para adecuarse a futuros requerimientos y problemas. Queremos evitar el rediseño, o minimizarlo.

No hay que resolver cada problema partiendo de cero, hay que reutilizar soluciones que ya han sido útiles en el pasado. Los patrones resuelven estos problemas concretos y hacen que los diseños OO sean más flexibles, elegantes y reutilizables.

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

Los patrones de diseño son más puntuales que los arquitectónicos, son de más bajo nivel, rozan la implementación.

No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular.

Los patrones se aplican (son como plantillas de solución genérica) a un problema particular, las soluciones que se obtienen es distinto cada vez que se aplica.

Patrones vs Idiomas

La diferencia de ambos es el alcance.

Un patrón de diseño tanto el problema como la solución son suficientemente genéricos, por lo tanto, son independientes del lenguaje de programación. En cambio, el Idioma es un patrón de bajo nivel ya que es específico a un lenguaje de programación.

A estos patrones si se los adapta, presentan para un lenguaje de programación particular se los llama Idioma.

¿Por qué utilizar patrones de diseño? porque te enseña a resolver todo tipo de problemas utilizando principios del diseño orientado a objetos.

- **Facilita la comunicación** en el equipo de trabajo, ya que con el nombre (si el equipo lo conoce) identificamos el problema a resolver.
- **Facilitan la reutilización** de buenos diseños y arquitecturas.
- **Nos ayudan a elegir alternativas de diseño** que hacen que un sistema sea reutilizable, y a evitar fallas que dificultan dicha reutilización.
- Pueden **mejorar la documentación y mantenimiento** de sistemas existentes.
- Ayudan a lograr un **buen diseño** rápidamente (mejoran la calidad de diseño).

¿Cómo seleccionar un patrón?

Debemos primero conocer los patrones, el nombre y el problema que intenta resolver.
Leer el propósito del patrón, las consecuencias (positivas y negativas) y en base de eso verificar si es conveniente aplicar el patrón.

¿Cómo usar un patrón?

- Se lee el patrón, se analiza.
- Clases participantes: se crean, se relacionan y qué métodos se necesitan agregar
- Se estudia la estructura, participantes y colaboraciones
- Cambios sobre el modelo de análisis obtenido en la etapa de análisis, rediseñando para mejorar la calidad

de diseño en términos de cohesión, acoplamiento, flexibilidad, mantenibilidad, etc.

Plantilla de definición

- Nombre
- Propósito
- Sinonimo
- Motivación
- Aplicabilidad
- Estructura
- Participantes
- Colaboraciones

Elementos

1. **Nombre**: describe con una o dos palabras un problema de diseño junto con sus soluciones y consecuencias.
2. **Problema**: describe cuándo aplicar el patrón. Explica el problema y su contexto.
3. **Solución**: describe los elementos que forman el diseño, sus relaciones, responsabilidades y colaboraciones. Es como una plantilla que puede aplicarse en muchas situaciones diferentes.
4. **Consecuencias**: son los resultados, las ventajas e inconvenientes de aplicar el patrón. Son críticas para evaluar el patrón y entender los costos y beneficios de aplicarlo.
   Una de las consecuencias que más se repite de aplicar patrones, es que como ventaja obtenemos Alta Cohesión (Cohesión funcional) cada clase hace una sola cosa (principio Single responsibility - Responsabilidad Única) y como desventaja se agregan más clases por ende crece la estructura.

¿A qué ayudan los patrones?

1. ***Encontrar objetos apropiados:*** muchos objetos de un diseño proceden del modelo de análisis, pero los diseños OO suelen acabar teniendo clases que no tienen su equivalente en el mundo real. Los patrones de diseño 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. ***Strategy y State.***
   Clases de fabricación pura (las q te da el patrón), clases que forman parte del dominio de la solución.

Buscamos clases de fabricación pura, clases de diseño, que forman parte del dominio de la solución. Son clases que agregamos nosotros que no tienen una traza con el dominio del problema porque no se pueden rastrear, son inventos nuestros. Esto nos ayuda a tener una alta cohesión, bajo acoplamiento, SW flexible y extensible. Cada patrón va a ir agregando clases que tengan que ver con la solución.

2. ***Determinar la granularidad de los objetos:*** Los patrones de diseño ayudan a decidir el nivel de detalle en la asignación de responsabilidades a los objetos, cuanto queremos que haga cada clase (cuántos métodos debería tener), apuntando siempre hacia una granularidad fina en busca de clases con alta cohesión. Relacionado con el principio SRP, modelamos una clase un concepto, que tenga una única razón para cambiar.
   Facade, Flyweight, Builder.
   Granularidad fina => Alta Cohesión (Responsabilidad mínima a las clases)
   Granularidad gruesa => Baja cohesión (Las clases hacen muchas cosas)

3. ***Especificar interfaces de objetos:*** La interfaz de un objeto caracteriza al conjunto completo de peticiones, identificadas por sus signaturas (nombre, parámetros, retorno y tipo), que se pueden enviar al objeto. Los objetos no se comunican entre sí, sólo se conocen a través de su interfaz. La interfaz de un objeto no dice nada acerca de su implementación (dos objetos con implementaciones diferentes pueden tener interfaces idénticas). Cuando se envía una petición a un objeto, la operación concreta que se ejecuta depende tanto de la petición

como del objeto que la recibe, ya que objetos diferentes que soportan peticiones con signaturas idénticas pueden tener distintas implementaciones en las operaciones que satisfacen esas peticiones, permitiendo sustituir objetos en tiempo de ejecución por otros que tengan la misma interfaz (POLIMORFISMO). Los patrones de diseño ayudan a definir interfaces identificando sus elementos clave y los tipos de datos que se envían a la interfaz, así como también puede decir que NO debemos poner en la interfaz. También especifican relaciones entre interfaces. ***Memento, Decorator, Proxy.***

Explicación Meles:
Primero definir una interfaz, que es……. Tiene que ver con el principio de programar hacia la interfaz y no hacia la implementación, nos conviene que las dependencias se hagan con abstracciones y no con concreciones. Los patrones nos ayudan a que aparezcan interfaces que nos permiten manejar el nivel de acoplamiento y manejar la transparencia para los clientes, sabiendo con quien se tiene que comunicar. Los patrones me ayudan a establecer lineamientos que cumplan con la segregación de interfaces.

Especificamos interfaces para que el cliente deje de comunicarse con cada clase concreta necesitando algo, entonces a través de una interfaz se mejora el nivel de acoplamiento comunicándose sólo con ella.
Es que los patrones ayudan a identificar interfaces para manejar el nivel de acoplamiento y manejar la transparencia hacia el cliente, mejorarle la vida al cliente para simplificar a quien debe comunicarse (interfaz).
Realización => Ventaja de que se resuelve dinámicamente (tiempo de ejecución), no rompe encapsulamiento, no rehusó codigo (métodos vacíos)

4. ***Especificar implementaciones de objetos:*** la implementación de un objeto queda definida por su clase. La clase específica los datos, la representación interna del objeto y define las operaciones que puede realizar. Por el contrario, el tipo de un objeto sólo se refiere a su interfaz, es decir, el conjunto de peticiones a las cuales puede responder. Puesto que una clase define las operaciones que puede realizar un objeto, también define el tipo del objeto. Manipular los objetos solamente en términos de la interfaz definida por las clases abstractas tiene 2 ventajas:
   1. Los clientes no tienen que conocer los tipos específicos de los objetos que usan, basta con que éstos se adhieran a la interfaz que esperan los clientes.
   2. Los clientes desconocen las clases que implementan dichos objetos.

   Explicación Meles:
   Cuando hablamos de implementación hablamos de código que va dentro de los métodos, se relaciona con el patrón state, ayuda con la especificación de implementaciones, cuando definis todos los comportamientos en la clase padre, definimos como se va a implementar determinado comportamiento que después los estados concretos van a tomar y lo van a tomar tal cual o lo redefinen.
   Hay otros patrones que definen lineamientos de cómo va a ser el algoritmo que se va a implementar.
   Ej: TM me define una forma de escribir un algoritmo que todas las clases hijas tienen q respetar la estructura de herencia para que el método plantilla funcione.

5. ***Favorecer reutilización:*** *Tiene que ver con el principio de favorecer la composición por sobre la herencia,* las dos técnicas más comunes para reutilizar funcionalidad en sistemas OO son el **reuso de caja blanca y reuso de caja negra**, las relaciones de clases son la herencia de clases/interfaces y la composición (asociación, agregación) de objetos.
   Herencia: rompe el encapsulamiento para que las clases hijas tomen estructura y comportamiento del padre, hay una dependencia muy fuerte, relación entre clases, no se traslada a los objetos, en tiempo de compilación, la solución sea mas rigida, menos flexible

Composición: relación de todo a parte, se define en tiempo de ejecución se traslada a los objetos, no rompe el encapsulamiento, sw más flexible. Hace reuso de caja negra a través de la delegación. Delegación es aprovechar el código que está escrito en otra clase, sin tener que romper el encapsulamiento y sin tener que escribir el código en la clase contexto, invoca al método para que otra clase lo resuelva.
Una clase tiene chances de ser reutilizada si tiene un bajo acoplamiento, cohesión funcional.
Relacionado con el principio DRY, TDA

6. ***Diseñar para el cambio:*** la clave para maximizar la reutilización reside en anticipar nuevos requerimientos y cambios en los existentes, y en diseñar los sistemas de manera que puedan evolucionar en consecuencia. Un diseño que no tenga en cuenta el cambio sufre el riesgo de tener que ser rediseñado por completo en el futuro. Cada patrón de diseño deja que algún aspecto de la estructura del sistema varíe independientemente de los otros, haciendo así al sistema más robusto y flexible frente a un tipo de cambio concreto.

Clasificación:

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

Los patrones se clasifican según su propósito en creación, estructura y comportamiento, de acuerdo al ámbito donde trabajan puede ser a nivel de clases (en tiempo de compilación, es estático) u objetos (es una relación más sana)

Patrones de creación:

Necesidad de creación de objetos de una clase. Separa el proceso de creación de objetos de su uso y representación. Su objetivo es abstraer el proceso de creación y ocultar los detalles de cómo los objetos son creados, mejorando y flexibilizando tal proceso a través de la **distribución de responsabilidades**. Todo lo que el sistema conoce de los objetos son sus interfaces. Nos permiten manejar el que se crea, cómo se crea, quién lo crea y cuándo se crea.

* Un patrón de creación de clases usa la HERENCIA para cambiar la clase que es instanciada.
* Un patrón de creación de objetos DELEGA la creación de la instancia a otro objeto. (Composición)

¿A qué ayuda?

Separar la clase que invoca el constructor de creación con la parte de cómo se va a crear, quién y cuándo se va a crear.
Flexibilizar el proceso de creación (**qué** se crea, **quién**, **cómo** y **cuándo** se crea), en Distribuir responsabilidades en distintas clases que se van a hacer cargo de cosas distintas (No tener una clase de que se haga cargo de la creación o de todo)

- ***Builder:*** Abstrae el proceso de creación de un objeto complejo (tiene muchas partes), centralizando dicho proceso en un único punto. Se utiliza cuando sabemos qué necesitamos crear, pero es necesario crearlo por partes y tener el control del proceso de creación.

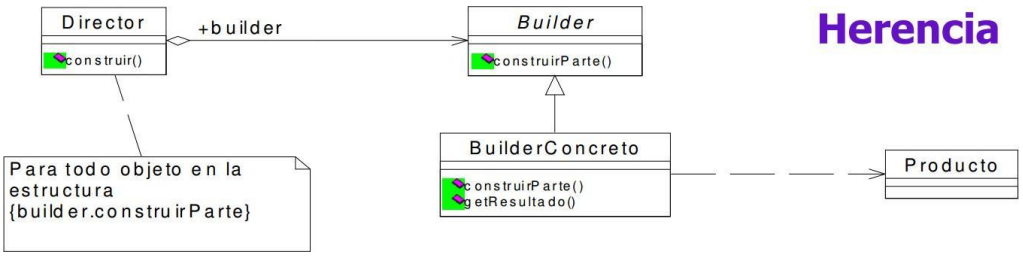
  Se usa para la creación de objetos complejos, que tienen muchas partes (objetos compuestos) y que se mantiene un control en el proceso de creación (paso1, paso2, paso3, etc). El resultado es un objeto complejo, compuesto por partes.

  Se hace una separación de intereses (el objeto que va a crear, el que lo usa, el momento de creación y responsable de creación), donde se distribuyen responsabilidades entre las clases de fabricación pura. Ejemplos de objetos complejos: Pantallas, reportes, estadísticas, etc.

  Director (Clase de fabricación pura) responsable de tener la lógica en el orden de como crear por parte y en dirigir el proceso de construcción, ***da órdenes***.
  Constructor Concreto (ya existía) es el responsable de crear los objetos productos, ***el que hace*** (el que trabaja).

  El *actor* elige el escenario y sabe que ConstructorConcreto va a usar en la dinámica y lo crea. Cada constructor concreto es el responsable de crear su producto concreto que es el que crea al final del proceso de creación.
  El contexto tiene una dependencia con el IConstructor para crear el constructor Concreto.
  El contexto siempre delega la responsabilidad al Director de crear pasandole la lógica. 1ro Se crea el constructor Concreto y se lo pasa por parámetro al director cuando se lo crea, para que el director dirija el proceso de construcción.
  Los constructores concretos implementan polimórficamente los métodos que están definidos en el IConstructor. Una vez finalizado el proceso de construcción, el contexto le pide al Constructor todo lo que creo, así lo puede utilizar.

  Propósito: Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.



Principios de diseño OO se aplican en el patrón Builder()

Responsabilidad Única (RSP):Cada constructor concreto va a tener la responsabilidad de crear un único producto

Programar hacia la interfaz (PTI):Desacoplar la dependencia haciendo que un director se comunique hacia una interfaz y no hacia los constructores concretos.

Open-Close(O/C): Si queremos agregar constructores debemos implementar la interfaz constructor.

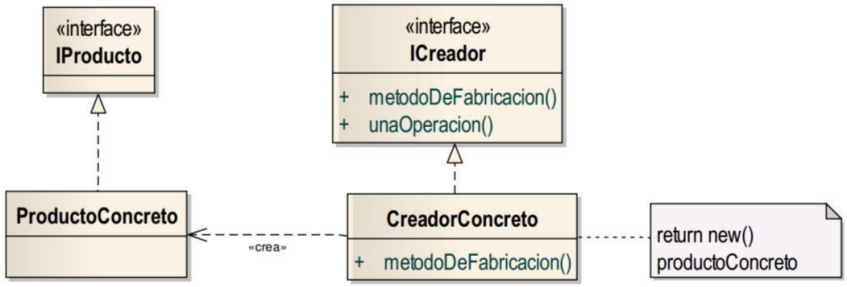
Ordena no pregunte(TDA): El contexto le ordena al director el proceso de creación de un objeto.

- ***Factory Method***: Centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la diversidad de casos particulares que se pueden prever, para elegir el subtipo que crear.
  (Define una interfaz para crear un objeto, pero deja que la subclase decidan qué clase instanciar. Delega en las subclases la creación de objeto)

  Se puede usar en Herencia y Realización. Tiene la libertad o flexibilidad de postergar lo que más se pueda, la decisión de cuál es el producto concreto que tiene que crear, entonces se especializa el método de construcción de acuerdo al producto concreto que tiene que construir.

  El **producto concreto** necesita de una interfaz común (a Diferencia del Builder), para todos los productos concretos, que se va a especializar según lo que necesitemos. El Builder crea objetos complejos a diferencia del Factory Method que crea objetos simples.

  MetodoDeFabricacion() -> (Como template ya que el patrón se aplica a un MÉTODO!) Es un método genérico, en el cual lo implementa cada creador Concreto donde crea su producto específico (Por eso existe la dependencia del creador concreto y el producto que crea)



  El método tiene toda la información para crear, y el retorno del método es el objeto creado. El contexto se puede convertir en la interfaz Creador.

Patrones de estructura:

Nos ayudan a combinar estructuras de objetos y clases complejas, manteniéndose flexibles y eficientes.
Resuelven 2 problemas:

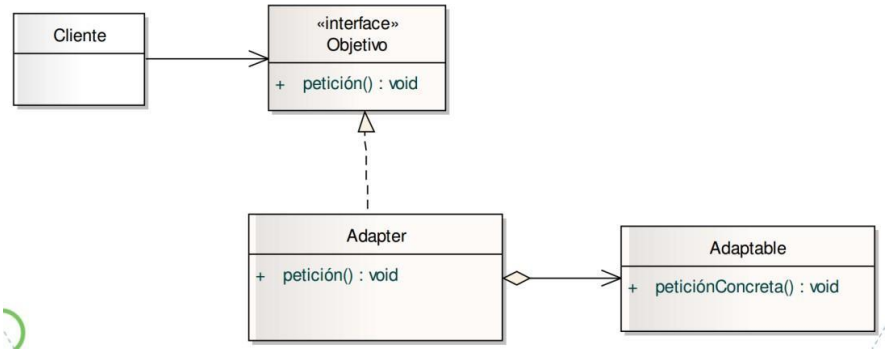
1. **Simplificar estructuras muy complejas**, resuelve necesidades puntuales de comportamiento en los objetos pero que se pueda hacer dinámicamente, en tiempo de ejecución. Si o si usamos relaciones que se trasladen a los objetos, asociación, agregación, composición. (Composite)
2. **Lograr compatibilidad entre clases incompatibles**, cuando necesitamos que una clase acceda a servicios o beneficios de otra clase pero no se ven. Se resuelve con la indirección (poner una clase en el medio que funcione como intermediario, haciendo que las 2 clases sean compatibles en términos de comunicación). (Adapter)

   * un patrón estructural de clases hace uso de la HERENCIA para componer interfaces o implementaciones.
   * un patrón estructural de objetos usan la COMPOSICIÓN para describir formas de componer objetos para obtener nueva funcionalidad.

Indirección -> Significa en términos generales poner una clase en el medio que funcione como intermediario, intérprete o traductor dependiendo el problema que sea. Hace que las dos clases sean compatibles.

Incorporar funcionalidad a objetos (esto se da en tiempo de ejecución)

- ***Adapter:*** Se utiliza el patrón cuando existen clases incompatibles, adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
  (Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permite que cooperen las clases que de otras maneras tendrían interfaces incompatibles.
  Intermediario entre dos clases que no pueden comunicarse



Principios de diseño OO se aplican en el patrón Adapter()

Segregación de interfaces: la interfaz intermedia tiene los comportameintos que le van a servir al cliente y los que tiene que implementar el adaptador.

Principio de responsabilidad única: Puedes separar la interfaz o el código de conversión de datos de la lógica de negocio primaria del programa.

Principio de abierto/cerrado: Puedes introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente.

- ***Composite:*** Permite tratar objetos compuestos como si fueran simples (Individuales).

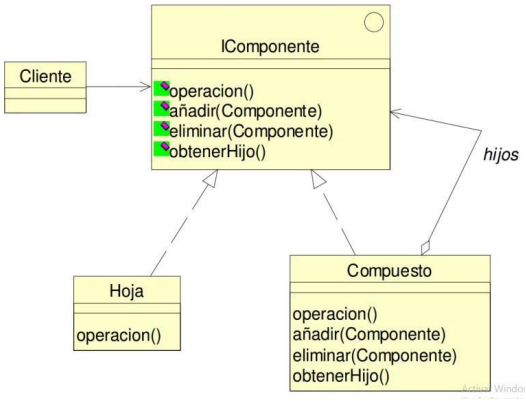
  Combina objetos en estructuras de árbol para representar jerarquías de todo-parte (Objetos individuales y compuestos)

  Debe existir una relación **jerarquía de tipo TODO-PARTE** (Agregación o Composición), que tenga a los sumo 3 niveles o sea recursiva para poder aplicar el patrón.

  Motivación: Simplifica estructuras complejas, para poder relacionarse con esa jerarquía (cliente que necesita acceder a esos servicios) y resuelve incompatibilidades entre clases.

  Son dos clases concretas que implementan la interfaz componente (IComponente). Tiene la facultad de cambiar la estructura base que viene del análisis, porque reemplaza clases por otras. Se debe rediseñar la estructura.

  Añadir, eliminar y obtenerHijo son métodos del patrón, son polimórficos. El cliente hace las peticiones hacia la interfaz.



Principios de diseño OO se aplican en el patrón Composite()

Programar hacia la interfaz y no hacia la implementación: Hago más transparente la interfaz para el cliente, decido que me preocupa más la interfaz hacia el cliente que la forma que voy a implementar una determinada solución.

Principio de abierto/cerrado: Puedes introducir nuevos tipos de elementos en la aplicación sin descomponer el código existente, que ahora funciona con el árbol de objetos.

El **composite no aplica** el principio de diseño:

Segregación de interfaces: xq define una interfaz que solo son aplicables para el compuesto y no para la hoja. Los declara en la interfaz xq prefiere privilegiar el servicio que ofrece al cliente, que sea más transparente con quien se relaciona. La hoja hace una implementación trivial

Rompe un principio de diseño -> *segregación de interfaz* (No obligues a las clases concretas a implementar comportamiento que no debería implementar)

Hijo, hace una implementación trivial (Esto no lo podes hacer, o no hagas nada, o no podes agregar componentes), hace eso ya que el método no puede quedar vacío (NO PUEDEN EXISTIR MÉTODOS VACÍOS EN CLASES CONCRETAS)

Una solución sería que el cliente tenga un puntero a la hoja y al compuesto (Pero aumenta acoplamiento, y rompe principio programar hacia la interfaz)

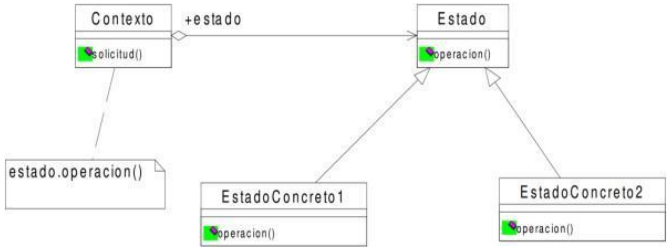
Deja de lado el principio segregación de interfaz para darle más lugar al programar hacia la interfaz, detrás de este principio está el concepto de transparencia para el cliente, se le da cierta información al cliente a través del composite, programar hacia la interfaz dice mejor pensar en el bienestar del cliente

Patrones de comportamiento:

Permiten manejar la comunicación entre objetos del sistema y el flujo de información entre ellos. Se encargan de distribuir y delegar la responsabilidad entre los objetos. Su objetivo es flexibilizar la relación entre los objetos.
Escribir algoritmos eficientes.

* un patrón de comportamiento basado en clases usa la HERENCIA para distribuir el comportamiento entre clases.
* un patrón de comportamiento basado en objetos usa la COMPOSICIÓN de objetos

- ***State:*** Tiene un solo estado en un momento determinado del tiempo. Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.
  Se usa cuando tengo clases que dependen del estado en el que se encuentra, por lo cual delega el comportamiento a la estructura del estado. Multiplicidad 1 a 1
  Resuelve el problema del switch case, if a través del polimorfismo, la desventaja es que se genera un gran crecimiento de clases, o sea una clase por cada estado.
  El contexto es el que delega la responsabilidad a la clase Estado abstracto (Invoca el mismo comportamiento en la clase Estado) , es decir, se relaciona con la abstracción e instancia un objeto concreto.
  El estado concreto es el que hace reutilizacion de codigo o lo redefine, todo el código está definido en la clase Estado abstracta, por eso el State funciona con la **Herencia**.
  Un método de estado en la clase contexto (ej cancelar()) invoca el comportamiento que está definido en la clase estado con un método (estado.cancelar()), pasándose a sí mismo (el objeto contexto) como parámetro para saber su estado actual (apuntando a un estado concreto). Si no se pasa a sí mismo como parámetro le tiene que pasar la información al estado para que lo pueda resolver. Le delega la responsabilidad al objeto en el que se encuentra (estado actual)



Principios de diseño OO se aplican en el patrón State ()

SRP: se reduce la responsabilidad en el Contexto y se crea cada Estado Concreto para mantener una sola responsabilidad

OCP: Si se necesita cambiar se agregan nuevos estados Concretos (se extiende) para manejar la variabilidad y comportamiento. Se crea una jerarquía de clases nuevas para no modificar la clase contexto que es la clase original.

LSP: los Concretos se pueden intercambiar con el Abstracto, ya que todos son capaces de responder a sus métodos.

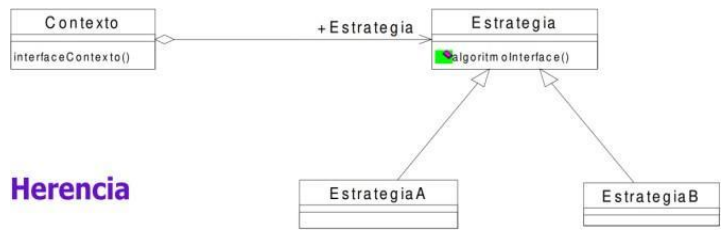
DRY: el código común se mantiene en la implementación trivial del padre, evitando redefinirlo en todas las hijas.

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:*** Se define una familia de algoritmos, en el cual encapsula cada uno de ellos en clases y elegir cuál estrategia utilizar en tiempo de ejecución.
  Se puede resolver con herencia y realización. *Alguien le informa al contexto la información específica para ver qué estrategia utilizar en ese momento* (actor que selecciona algo, por parámetro, etc), el contexto puede tener más de una estrategia habilitada al mismo tiempo en la estructura (Multiplicidad 1 a *).

  El contexto le delega a la estrategia. *En herencia* cada estrategia concreta implementa lo que heredó de la estrategia abstracta, *en realización* las estrategias concretas implementan el comportamiento polimórfico que está declarado en la interfaz como un método vacío. Las clases concretas no pueden tener métodos vacíos. Conviene utilizar realización ya que cada estrategia concreta es diferente a la otra y no hay reutilizacion de codigo.
  El contexto crea la interfaz concreta con un new() y se pasa como parámetro o pasa la información para que la estrategia pueda resolver el problema.
  La delegación está en que el contexto tiene una invocación al método de la Estrategia.



Principios de diseño OO que se aplican en el Strategy ()

OCP: el Contexto está abierto para extensión, ya que se pueden agregar más Estrategias. [ok]

ISP: Le da a las clases concretas los métodos que necesita implementar, cada estrategia es específica (alta cohesión). [ok]

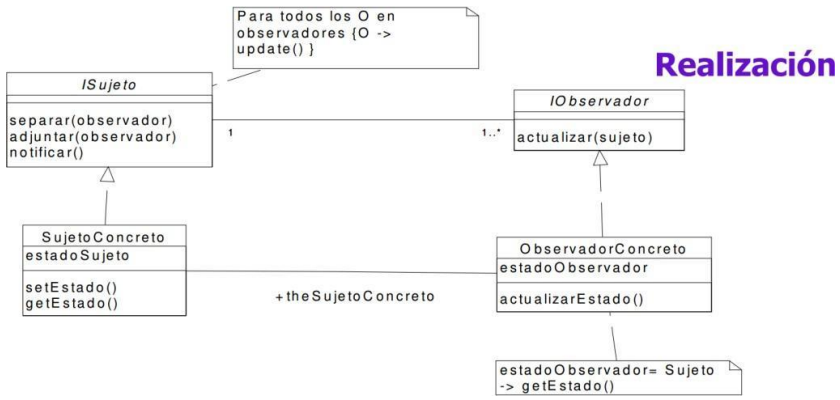
DRY: la parte común del algoritmo se mantiene en el contexto, evitando duplicaciones. El contexto tiene la llamada del método, a través de la delegación el código de ese método lo tiene la estrategia que lo resuelve.

ECV: la parte que varía del algoritmo se encapsula en las Estrategias. Ej: cálculos que saquen el iva (puede cambiar el iva)

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 referencias a las Estrategias, en vez de una clase base con el comportamiento general e hijas con la parte que cambia. Se analiza el contexto con la estrategia. Aplicamos delegación en contexto y estrategia, reutilizamos código xq solo hacemos la llamada.

- ***Observer:*** Define una dependencia de uno-a-muchos entre objetos, de forma que *cuando cambia el **estado** un objeto (Sujeto), todos sus dependientes (Observadores) automáticamente son notificados y actualizados*.
  Trabajar con actualización de componentes, que el ámbito son los observadores, frente al cambio de estado del sujeto. La estructura puede ser con Herencia o Realización.
  Nivel bajo o casi nulo de acoplamiento, entre los observadores y sujeto. Están desacoplados, se pueden agregar o quitar de forma fácil.
  Un ejemplo puede ser el Modelo - Vista, donde el modelo tiene los datos y se modifican, tiene que notificar y actualizar a los observadores que son las vistas.
  Se puede aplicar este patrón con la forma Pull (se lo notifica y se actualiza cuando lo desea, es decir busca los datos que necesita). En cambio el Push (se pasa la información forzosamente para que actualice)



Principios de diseño OO se aplican en el patrón Observer()

Segregación de interfaces (ISP): Tengo 2 interfaces, pongo el comportamiento que tiene q ver con los sujetos en la ISujeto para que los sujetos concretos lo implementen y con el observador igual.
Programar hacia la interfaz y no hacia la implementación (PTI): reducimos el acoplamiento entre sujetos y observadores xq se comunican a través de la interfaz.
Abierto para la extensión y cerrado para la modificación (OCP): Puedo introducir nuevas clases observadores sin tener que modificar el sujeto (solamente implementando la interfaz de observadores) y viceversa

- ***Template Method:*** Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.

  Si o si herencia (si no hay, no hay template method). Se aplica a nivel de 1 método, llamado método plantilla, entonces el propósito es garantizar que un algoritmo se va a ejecutar exactamente en orden que yo quiero que se ejecuten las operaciones primitivas que yo tengo que ejecutar.

  Este patrón se usa cuando queremos garantizar que todas las subclases que van a heredar el método plantilla, implementan el algoritmo que está contenido, en el orden de invocación de las operaciones que está definido en el método.

  Estructura el orden del algoritmo, dentro del algoritmo se ejecutan una serie de pasos (hacer esto, después esto, etc.). Ej liquidación de sueldo (1ro conceptos del sueldo bruto, 2do deducciones, 3ro aportes voluntarios..)

  El orden es el mismo para todas las subclases del método plantilla implementado (el que lleva el orden), pero cada subclase dentro de cada operación hace algo distinto.
  Tengo que garantizar la estructura de ejecución de las operaciones (Proteger con método plantilla), pero le doy libertad a cada clase concreta de que las operaciones las resuelva de manera diferente.

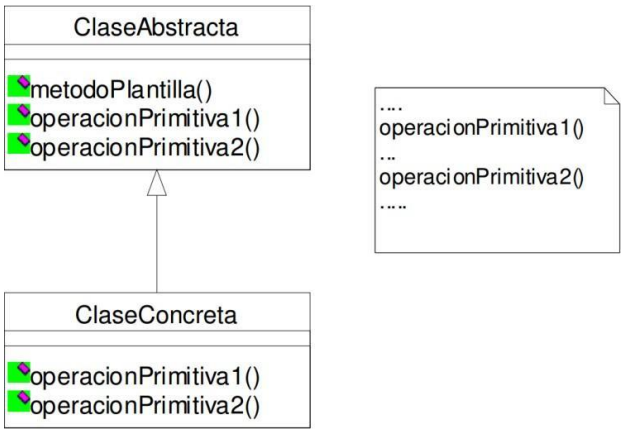
  El método plantilla se define en la clase PADRE como público final, para que las hijas no lo puedan modificar, pero sí modificar polimórficamente las operaciones que se invocan desde el método plantilla.

  Si no hay un algoritmo con n° operaciones que invocar para resolver un problema con un orden en particular, no hace falta utilizar template.

  Un método plantilla (template) invoca los siguientes tipos de métodos:
  - Operaciones abstractas del padre.
  - Operaciones concretas del padre.
  - Invocando, operaciones concretas de los hijos.
  - Método hook (Método enganche), lo tiene definido el padre para que todos los hijos lo puedan usar o ignorar en función de lo que necesiten (es un comportamiento por defecto). También puede ser vacío.

  Estructura de control invertida, relacionada con el principio Inversión de dependencia
  Las invocaciones en la herencia son del hijo al padre (hijo --I> padre), principalmente los hijos invocan operaciones del padre, acá se invierte el orden natural. Se plantea que dentro del metodoPlantilla() que está en el PADRE, tengo invocaciones a las operaciones concretas de las clases hijas.
  Principio de hollywood (No nos llamen, nosotros te llamamos) -> La clase padre llama a la clase hija. Se da en tiempo de compilación.

  En términos de objetos, la clase concreta (Hija) es instanciada y hereda el método plantilla, invocando la operación a sí misma. Ya que la clase abstracta no se puede instanciar. **En términos de estructura** estamos aplicando la inversión de dependencia, porque estamos haciendo una invocación de un método de la clase hijo en el método plantilla que está en el padre.



En la clase concreta no se ve el método plantilla xq solo se visualiza lo que se redefine. Primero se implementan los métodos de enganche y luego los que se redefinen.

Principios de diseño OO se aplican en el patrón Template Method()

LSP: Analiza si está bien armada la herencia. (hay q mejorarlo pero lo cumple)

DIP: principio de inversión de dependencia. Le da más potencia al método plantilla y que permite que en el método plantilla haya declarado invocaciones a sus clases hijas (naturalmente la herencia no lo permite), aca el padre apunta al hijo y en herencia es al revés. Permite que el TM pueda armar una plantilla completa, con invocaciones a clases concretas.

PRINCIPIOS DE DISEÑO

Principio de diseño: es una técnica o herramienta básica que se aplica para construir SW y que nos permiten evitar tener un mal diseño, obteniendo SW más flexible, extensible y mantenible. Son guías de alto nivel, que 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*.

- *DRY (Don't Repeat Yourself - No te repitas)*: busca que cada funcionalidad del sistema esté representada por una única porción de código. Así el SW se puede modificar sin romper partes no relacionadas al cambio, y se mantiene consistente al no correr el riesgo de que la misma funcionalidad se haga de dos maneras diferentes.

  Patrón que aplica: State, porque todo el código está definido en la clase abstracta

  Evitar duplicaciones de código repetido con el mismo objetivo, definiéndolos en un único lugar, de forma tal que creamos abstracciones. De esta forma logro que *cada requerimiento esté en un único lugar, representada por una única porción de código*. Nos permite no repetir código o si algo cambia en el SW solo modificar en un único lugar sin que repercuta en todo el SW. Ubicar la pieza de información en un único lugar y referenciarla desde donde la necesitemos, evita inconsistencias, evita información duplicada.

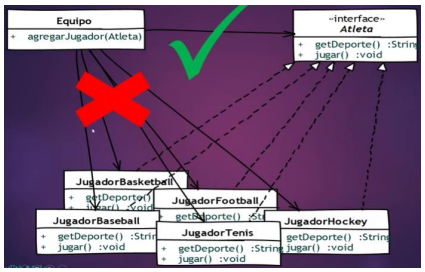
- *EWV (Encapsulate What Varies - Encapsula lo que varía)*: También conocido como principio SOLID Open Closed, busca separar del código las partes propensas a cambiar y encapsularlas en otros objetos, para que cada vez que haya que hacer un cambio el impacto está localizado en una pequeña y única parte del sistema. Mejora la flexibilidad y mantenibilidad del sistema y permite manejar la variabilidad. Busca agregar clases nuevas que soporten la variabilidad de los requerimientos que podemos tener sobre un determinado comportamiento de una clase.

  Patrón que aplica: **State, strategy, una clase que encapsula el código que puede llegar a cambiar**

- *PTI (Programming To Interfaces - Programar hacia la interfaz):* *programar hacia la interfaz y no hacia la implementación*. 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. Se define primero la interfaz o contrato que el módulo deberá proveer, y luego se implementa la funcionalidad necesaria para realizar ese contrato. Así no se expone la funcionalidad interna y el módulo tiene un contrato claro de qué es lo que puede hacer. Busca la transparencia con el cliente, hacerle la vida más fácil.

  Patrón Strategy: aquí se aplica al establecer una interfaz única, donde todas las Estrategias deben ser implementadas hacia ella para poder ser utilizadas en el Contexto. Nos interesa saber QUE hace, no COMO lo hace. La interfaz está definida por su signatura (nombre, parámetro, retorno y tipo), los métodos de una interfaz son vacíos. La interfaz define un contrato, el contrato que define son los métodos, o sea cualquiera que realice la interfaz tiene que poder responder a esos métodos.
  El contrato se establece entre la interfaz y las clases concretas que la implementan (las que definen el CÓMO se hace, el código).

  Las interfaces me aseguran que un objeto va a poder realizar una acción, entonces cualquier cliente que necesite de esa acción va a poder pedírselo a través de esa interfaz. **Cada clase que implemente una interfaz debe cumplir con los comportamientos de la misma.**



- *COI (Composition Over Inheritance - Composición por sobre herencia): favorecer la composición sobre la herencia*. La herencia genera fuerte acoplamiento entre las clases hijas y el padre, y rompe el encapsulamiento, ya que ellas acceden a los atributos del padre. Se propone usar composición, que nos permite reutilizar comportamiento (código) a través de la delegación, manteniendo un acoplamiento menor, y además el cliente no conoce el interior de la clase contenida, respetando el encapsulamiento. Usar la delegación como reuso en vez de utilizar la herencia. En el caso de la realización estoy reusando la signatura del método.
  A través de la delegación usamos polimorfismo, dado que las clases están encapsuladas y pueden tener el mismo método, misma signatura, y los objetos se van a comportar de acuerdo a su clase, no metiendome en la estructura interna del código (así funciona la composición, con una llamada al método que necesita).
  Beneficios: Se resuelve en tiempo de ejecución, se define dinámicamente, no rompe encapsulamiento y el comportamiento depende de las relaciones.

  Patrón que aplica: Strategy

- *TDA (Tell Don't Ask - Ordena, no preguntes)*: El paradigma OO se trata de objetos que colaboran entre sí para implementar un comportamiento que solos no podrían, y el principio fundamental de esa colaboración es la distribución de responsabilidades, entonces un objeto le pide a otro objeto que haga algo que necesita y le devuelva el resultado. Este principio habla de distribución de responsabilidades. No tener clases que solo sean get y set.
  Busca llevar el código hacia donde se encuentran los datos que utiliza. Si un objeto tiene los datos necesarios para realizar una función, mejor que sea el responsable. El beneficio es que mejora la cohesión de cada clase al encapsular el código con los datos que utiliza, hace que los objetos trabajen de una formas más equilibrada. Se relaciona con el *Patrón Experto*. Asigna la responsabilidad al objeto que tiene la información.
  No usar los objetos para pedirle cosas, sino que hay que decirle a los objetos que hagan cosas, un objeto se define por su comportamiento y en términos finales todo se trata de la distribución de responsabilidades

  Patrón que aplica: State, builder

Principios S.O.L.I.D:

SRP (Single responsibility - Responsabilidad Única): Modelamos una clase, un concepto, hace referencia a mantener los conceptos en un único lugar, si defino una clase que modele un único concepto, esa clase solo debe cambiar por una única razón. Un módulo, clase o subsistema debería tener una única responsabilidad en el sistema totalmente encapsulada en sí mismo. Debe tener un solo rol dentro del SW, relacionado con el concepto que esa clase modela que define su responsabilidad. Cada parte del SW debería tener una única razón para cambiar. Se busca tener una cohesión funcional.
única responsabilidad + única razón para cambiar = ***alta cohesión***

Patrón que aplica: State, strategy, builder, adapter

Puedo tener varios métodos relacionados con esa responsabilidad.
Responsabilidad: Obligación que tiene un objeto en cuanto a su comportamiento, donde implementamos métodos para

llevarla a cabo.

OCP (Open-Closed - Abierto-Cerrado): las entidades de SW deben estar cerradas para la modificación pero abiertas para la extensión de funcionalidad. Ante la necesidad de un cambio no debería cambiar el SW existente, sino agregar algo nuevo que trabaje con el anterior sin modificar su funcionamiento. Frente a la situación de cambio de requerimientos o de comportamientos variables, en lugar de abrir la clase para que hagan algo nuevo, creamos otra clase que soporte el nuevo comportamiento demandado. Extender significa agregar clases nuevas.
permitir el cambio + no modificar lo existente = flexibilidad

Patrón que aplica: State, strategy, observer, builder, adapter, composite

Ej: Tengo una clase pintor y quiero agregar un nuevo estilo de pintura, extiendo con clases nuevas para soportar el comportamiento y aplicando polimorfismo redefino el método pintar(), que es igual a todas las clases, entonces polimórficamente implemento el pintar segun que es lo que necesito.

LSP (Liskov substitution - Sustitución de Liskov): Habla de una herencia bien diseñada, para el reuso de comportamiento, surge para reutilizar código. *Los subtipos deberían poder reemplazarse por sus tipos base sin romper el sistema*, el hijo se debe comportar como el padre. Es una forma de control sobre el diseño, y busca asegurar que los subtipos (hijos) cumplan correctamente con las interfaces propuestas por sus tipos base (padre), ya que sus métodos reciben los mismos tipos de parámetros y tienen igual tipo de retorno.
El liskov propone una prueba conceptual: tomar el código que está en el padre y lo pongo en la clase hijo para ver si funciona sin desvirtuar su esencia, si funciona es xq la herencia está bien pensada.
Nos preguntamos si el hijo puede sustituir al padre, para saber si estamos aplicando una herencia que tenga sentido.
Se debe probar de sustituir el hijo por el padre y que siga cumpliendo con la misma funcionalidad (tomar su responsabilidad), se utilizar para no forzar herencia.
El beneficio que nos provee la herencia es la reutilización de comportamiento (código), la herencia es la materialización del principio no te repitas (DRY) y rehuso, de ser más efectivos, de ser más eficientes, trabajar menos. La herencia sana es la herencia de comportamiento no la de estructura.

Patrón que aplica: State, Template method

tipos sustituibles por sus tipos base + revela problemas de estructura si existieran = herencia bien diseñada
- Herencia bien diseñada => Padre e Hijo con Igual **comportamiento**, cuando no se pierde el objetivo
- χ Herencia mal diseñada => Padre e Hijo solo se parecen en atributos

Ej: Hay 2 robots padre e hijo, Si el padre hace café el hijo puede hacer un tipo de café, pero no hacer algo diferente a café.

ISP (Interface segregation - segregación de interfaz): Se relaciona una clase concreta con una interfaz, que la implementa. Se ve a la interfaz como un contrato con cada cliente, que debería ser lo más pequeño posible (pocos métodos), para que ese cliente acceda al comportamiento declarado en la interfaz. *El objetivo es no forzar a los clientes a interactuar o relacionarse con interfaces muy complejas, con cosas que no necesita (depender de métodos que no utiliza), ni forzar las clases concretas a implementar comportamiento que no necesitan o no pueden*. (Alta cohesión en interfaces). Una clase concreta puede implementar todas las interfaces que quiera. Entonces tenemos que dividir las interfaces y hacerlas altamente cohesivas.

Patrón que aplica: Strategy, observer, adapter

no obligar a los clientes a depender de métodos que no utilizan + no obligar a las clases a implementar interfaces que no necesitan = bajo acoplamiento, alta cohesión, facilidad de implementación y prueba
El principio dice 2 cosas:
- para el lado de la clase concreta, no la tengo que obligar a implementar comportamiento que no necesita.

- para el lado del cliente (el que hace las peticiones a la interfaz), si pongo mucho comportamiento en la interfaz se vuelve muy compleja, entonces para el cliente es mucho más complejo interactuar que si tuviese una interfaz sencilla. No obligarlos a depender de métodos que no necesitan.

DIP (Dependency inversion - Inversión de dependencia): se busca evitar que *los módulos de alto nivel dependan de los módulos de menor nivel, sino desacoplar poniendo una interfaz en el medio, entonces los módulos de bajo nivel proveen los servicios que requiere el módulo de alto nivel*, pudiendo los de bajo nivel intercambiarse según las peticiones del módulo de alto nivel. El módulo de alto nivel necesita un servicio y el de bajo nivel se lo provee, pero a través de una interfaz. Nos brinda flexibilidad y robustez. No hacer depender a las clases o componentes de concreciones sino de abstracciones.

Patrón que aplica: Template Method

Se desacoplan ambas clases y se puede reemplazar una implementación de bajo nivel sin afectar a las clases base que la utilicen. (Principio de Hollywood) (Programar hacia la interfaz y no hacia la implementación) Reduce el acoplamiento.
los módulos de alto nivel no dependen de los de menor nivel + ambos deben depender de sus abstracciones + las abstracciones no deben depender de los detalles= flexibilidad, robustez, movilidad

Apunta a que implementes una relación de manera inversa a lo que naturalmente se está acostumbrado a que funcione, ***esto nos permite manejar dependencias entre componentes de forma que no dependas de componentes que están en un nivel jerárquico más bajo, entonces para evitar esto (llamado indirección) agregamos una interfaz o algo en el medio que nos permita establecer ese vínculo de manera inversa que naturalmente se había pensado.***
Ej: La relación en el TM es del hijo al padre, en este caso para poder implementar la plantilla el padre apunta al hijo (invierte la dependencia). Se introduce este recurso para que funcione al revés de lo que funciona.
Ej: Esquema de persistencia: Cuando tengo comunicación entre capas, la relación natural es del cliente (presentación) hacia la base de datos (de arriba hacia abajo), en este caso para que el esquema funcione con un buen nivel de cohesión permitimos que el esquema de persistencia le haga una petición a la lógica de negocio (lo natural sería que la LN pida y la BD suba la rta)

Conceptos:

- *Interdependencia Funcional:* la funcionalidad de un componente debe ser cohesiva y centrarse en una y sólo una función. El concepto de interdependencia funcional es resultado de la separación de problemas y de los conceptos de abstracción y ocultamiento de información. Se logra desarrollando módulos donde cada uno resuelva un subconjunto específico de requerimientos y tenga una interfaz sencilla. El SW con modularidad eficaz, es decir, con módulos independientes, es más fácil de desarrollar porque su función se subdivide y las interfaces se simplifican. Los módulos independientes son más fáciles de mantener (y probar) debido a que los efectos secundarios causados por el diseño o por la modificación del código son limitados, se reduce la propagación del error y es posible obtener módulos reutilizables. La independencia se evalúa con el uso de dos criterios:
- ***Cohesión***: un módulo cohesivo ejecuta una sola tarea, por lo que requiere interactuar poco con otros componentes en otras partes del programa.
- ***Acoplamiento***: es un indicador de la independencia relativa entre módulos. En el diseño de SW, debe buscarse el mínimo acoplamiento posible.
- *Reusabilidad:* grado en el que un programa puede volverse a utilizar en otras aplicaciones (se relaciona con el empaque y el alcance de las funciones que lleva a cabo el programa).
- *Modularidad:* construir SW que tenga modularidad eficaz, cada módulo debe centrarse exclusivamente en un aspecto bien delimitado del sistema: debe ser cohesivo en su función o restringido en el contenido que representa. Los módulos deben estar interconectados sencillamente: poco acoplamiento. La modularidad es la

manifestación más común de la división de problemas. El SW se divide en componentes con nombres distintos y abordables por separado, llamados módulos, que se integran para satisfacer los requerimientos del problema.

Principios de diseño vinculados con el concepto de Cohesión: **S** – **I**; **T**DA- **E**WV

Delegación: *Significa pedirle a alguien que haga algo por mí, confiar y darle el 100% de la responsabilidad para que lo realice y me devuelva el resultado*. Mecanismo que me permite polimórficamente aprovechar un código que está escrito en otra clase y obtener el resultado sin tener que romper el encapsulamiento de esta clase(Caja negra), sin tener que escribir el mismo código, solo hago la invocación. **El contexto de relación que permite la *delegación* es** la asociación, agregación y composición. *Delegar la responsabilidad a un objeto es diferente a pedirle colaboración. Lo usan patrones como State, Strategy, composite, etc.*

Es lo que nos permite hacer reutilización con composición en lugar de herencia. Un objeto le delega una tarea a un delegado, resolviendo una petición entre 2 objetos o más. La idea es componer un objeto con instancias de otros, y permitirle que le pida a sus delegados que resuelvan cosas por él, logrando tener un "pseudo objeto" más complejo. La relación es sumamente flexible ya que los delegados pueden cambiar en tiempo de ejecución. Si se utilizan interfaces en lugar de implementaciones, el objeto contenedor no tiene por qué saber cómo se van a resolver sus peticiones, solo las delega y otro le entregará los resultados

Características	HERENCIA	REALIZACIÓN (Interfaces)	COMPOSICIÓN
Propósito	**Extender** una clase base con clases hijas que reutilicen su comportamiento y agreguen el propio.	Definir una interfaz (**contrato**) común, para que otras clases puedan cumplirlo y así unirse bajo un contrato común. Me define que se puede hacer, no cómo se hace	**Contener** clases dentro de otras clases, para que las contenedoras puedan usar los métodos expuestos por las contenidas.
Características	**- Relación fuerte** de generalización en donde un objeto especificado se basa en la especificación de otro (comparten código). **Una clase padre comparte sus métodos y atributos con sus clases hijas.** - Esta relación permite definir una implementación en términos de otra, lo que se conoce como **reutilización de caja blanca**, ya que la estructura interna de las clases es visible (viola encapsulamiento). - No requiere interfaz ya que puede acceder a los miembros del padre.	**- Relación débil** entre clasificadores en donde un objeto clasificador establece un contrato que garantiza que otro lo cumplirá. Se utiliza entre interfaces y colaboraciones. - **Se utiliza para representar la relación entre una interfaz y la clase que la implementa**. **Describe cuándo se puede usar un objeto en lugar de otro.** - Clases donde los métodos no tienen comportamiento, consiste solo en la firma (nombre, atributos, retorno, tipo) - No puede tener comportamiento, todos sus métodos son abstractos	- **Relación todo-parte**, a diferencia de la agregación, las partes no tienen vida independiente fuera del todo, cada parte pertenece a al menos un y sólo un todo. *En agregación una parte se puede compartir entre los todos.* - **La funcionalidad se obtiene componiendo objetos, requiriendo que estos tengan interfaces bien definidas**. Esto se conoce como **reutilización de caja negra**, porque los detalles internos de los objetos no son visibles. - Requiere que la clase tenga una **interfaz bien definida** para poder usarla.
Polimorfismo	Si	Si	No directamente por sí sola.
Tipo de Relación	es un	es un	tiene un
Se puede modificar en	Compilación (cuando programo). Trabaja con clases	Compilación	Ejecución (el sistema se ejecuta). Trabaja con Objetos

Permite reuso de código	Si	No	Si
Permite encapsulamiento	No	N/A	Si
Acoplamiento	Alto	Bajo	Medio/Bajo
Multiplicidad	1..1 En general solo se puede heredar una clase	1..*	1..* Se puede agregar muchas clases, obteniendo así múltiple reutilización
Ventajas	* Fácil de implementar, porque se tiene acceso al contenido de la clase padre. * Permite cambiar las *implementaciones* heredadas de las clases * Aumenta la reutilización de código	* Mucho más flexible y robusta que la herencia * Proporciona un mecanismo para establecer contratos	* Se define en tiempo de *ejecución (flexible)* * No rompe el encapsulamiento * Posee bajo acoplamiento * más consistente
Desventajas	* Rompe el encapsulamiento * Produce alto acoplamiento * No permite modificar las implementaciones en tiempo de ejecución, ya que se define en tiempo de *compilación (rígida)*	* No proporciona reutilización de código	* No es polimórfica por sí misma * Construir un sistema basado en composición es mucho más costoso y difícil de implementar * Más difícil de implementar porque trabaja únicamente con lo que la clase expone, a través de la delegación.

MAPEO DE ESTRUCTURAS DE CLASES A BASES DE DATOS RELACIONALES – PATRONES DE PERSISTENCIA:

MAPEO DE CLASES A BD RELACIONAL: resuelve parte del problema de impedancia

Objetos en tablas: primero se debe decidir qué clases y variables van a ser almacenadas en la BD: las *clases de negocio* son las candidatas a persistencia ya que "sobreviven" a los CU, en oposición a las de *fabricación pura*. Cada clase a persistir será representada por una tabla o más. Una clase es mapeada en una tabla de la siguiente manera:

1. Mirar nuestra estructura de clase y analizar qué clases necesitamos que sean persistentes.
2. Asignar una tabla para la clase. Necesitamos que la tabla se llame como la clase para mantener una trazabilidad
3. Crear una columna para cada atributo: si el atributo es complejo se añade otra tabla para ese atributo (ejemplo: Persona -> País), o se distribuye en varias.
4. Se añade un atributo ID al objeto que se corresponderá con la PK, con un método para generarlo asegurando su unicidad (así 1 objeto en el sistema se corresponde con un 1 objeto en la BD), que no sea visible al cliente. Cada atributo se transforma en una columna de la tabla.
5. Cada instancia de la clase será representada por una fila de la tabla en la BDR.
6. Cualquier relación con cardinalidad mayor a 1(0..N, *…*) se resuelve con una tabla intermedia que conecta las tablas mediante el ID de la FK. En algunos casos, se puede representar la relación como una columna (atributo) en la tabla del objeto asociado.

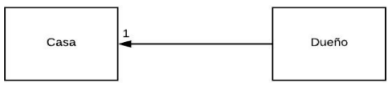
Normalización: consiste en el hecho de eliminar redundancia en la BD y evitar ciertas anomalías en la actualización. Un diseño de BD basado en un modelo de objetos acabará normalmente en 3FN desde el comienzo: si y solo si para todas las veces, cada fila consiste de un único objeto identificador junto con un número de valores de atributos mutuamente independientes, *la tabla está en 3FN*.

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

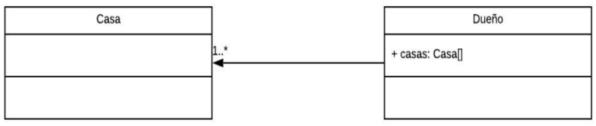
Hay 2 relaciones entre clases que requieren persistencia:

- ***Hay tres situaciones al mapear relaciones de asociación (agregación, composición):***
  1) *Relaciones 1 a 1:* se pide que un dueño tenga una casa y una casa tenga un dueño.

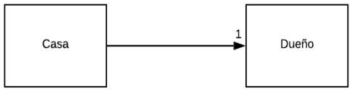
Esto se mapea como está, teniendo el dueño una FK de la casa en su tabla.



  2) *Relaciones 1 a *:* el cliente cambió los requerimientos y ahora los dueños pueden tener múltiples casas.

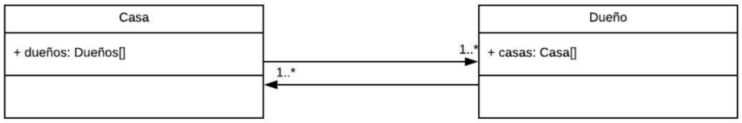


Damos vuelta la relación dejando que la casa tenga la FK del dueño.



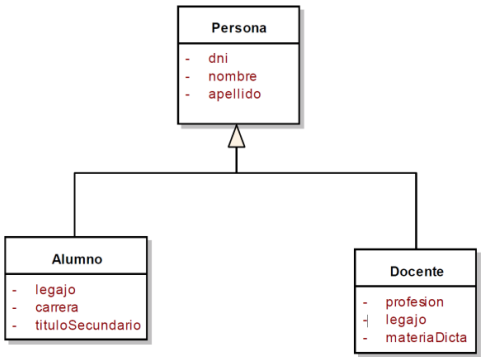
Autor: Andermatten – Martinez – Sánchez Díaz

3) *Relaciones * a *:* el cliente decidió que las casas ahora pueden tener múltiples dueños.

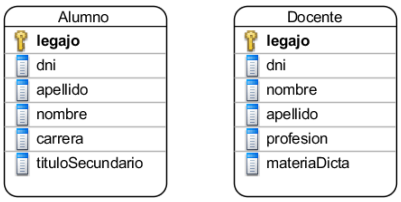


Generamos una tabla intermedia que tenga las FK de los dueños y casas.

❖ ***Hay 3 situaciones al mapear relaciones de herencia/generalización:***

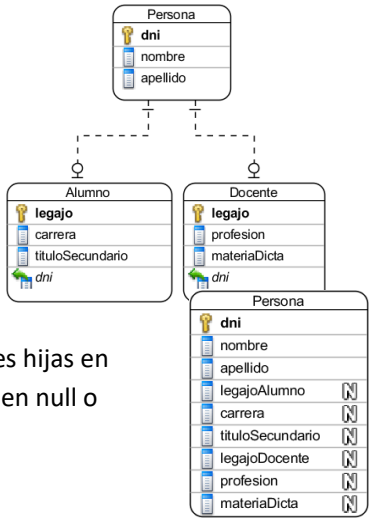


1) *Eliminar al padre:* se tienen tablas únicas para los objetos hijos. Se copian todos los atributos y relaciones de la clase padre y ninguna tabla representa la clase abstracta. Donde había Persona ahora puede haber un FK a Alumno o Docente. Siempre hay que mantener la consistencia en caso de que un Alumno sea también Docente.



- χ es más rápida ya que los datos están en una sola tabla: evita el uso de JOINs.
- χ el tamaño de la base de datos aumenta ya que se duplican columnas: aumenta el acoplamiento.
- χ puede traer problemas de consistencia y redundancia.

2) *Simular herencia:* la clase abstracta está en su propia tabla y las tablas de los hijos hacen referencia a ella.



- χ reduce la redundancia.
- χ siempre habrá que hacer JOINs, reduciendo la performance.

3) *Eliminar a los hijos:* uniéndolos al padre, se ponen los atributos de todas las clases hijas en una sola tabla. si se quiere almacenar un Alumno se dejan los datos del Docente en null o viceversa.

χ reduce altamente la cohesión. (no recomendado)

DISEÑO DE PERSISTENCIA

Implica analizar nuestras clases, definir cual de esas clases necesitan persistencia, en que almacenamiento persistente la 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).

Cuando necesitamos que nuestros objetos sean persistentes y no tenemos una BD OO, surge nuestra necesidad de buscarle una solución al paradigma.

Concepto de Objeto: Comportamiento (su rol, qué es lo que va a hacer en el dominio), estado (valor que pueden tomar los atributos), identidad (permite que sea único, depende de algo externo, es extrínseca, no depende del valor de los atributos). En el modelo de entidad relación la identidad si depende del valor del atributo

Problema de impedancia: ocurre cuando queremos mapear objetos (guardar objetos) a BD relacionales. 1er problema las BDR no soportan comportamiento, solo guardan datos. 2do problema, no puedo guardar cualquier tipo de datos, porque solo permiten primitivos, definidos por el motor.

Al trabajar en un lenguaje de programación OO, toda la información se almacena en objetos, por lo tanto, se necesita transformar la estructura de información de objetos a una estructura orientada a tablas (modelo relacional).

Problema de impedancia: es la dificultad que surge al utilizar una BDR para persistir objetos de un lenguaje OO. Se producen porque no existe una correspondencia directa entre estos dos sistemas, por lo que *hay que crear un esquema de correspondencia* (*mecanismos de mapeo*) para resolverlo. Se debe decidir la forma en que se almacenarán los atributos complejos de los objetos y las relaciones entre ellos.

Es el problema de guardar objetos que tienen estructuras muy complejas en un modelo relacional:

- Los objetos se componen de identidad, estado y comportamiento: las dos primeras (datos)se pueden almacenar en una BDR, pero *el comportamiento no*, por lo que se almacenan sólo el estado e identidad, y cuando se quiere recuperar el objeto se lo recompone con estos dos más el comportamiento de su clase definido en su estructura.
- Las BDR almacena la información en tablas que *sólo usan tipos primitivos* (int, byte, char, etc), y es normal que los objetos se compongan de más objetos: *se soluciona utilizando múltiples tablas, y relacionándolas con sus PK*.
- Se debe *violar el encapsulamiento* para extraer el estado del objeto: atributos que deberían ser privados e inaccesibles desde el exterior tienen que exponerse para almacenarlos en la BD.
- Las *relaciones de herencia y asociación no son soportadas* nativamente por las BDR y se deben implementar mecanismos para simularlas: encontrar una manera de representar la PK y muchas veces los objetos no tienen atributos que sirvan, por lo que se debe añadir uno y esconderlo de los clientes, ya que no es parte del objeto en sí, sino una herramienta para integrar con la BDR.
- Genera *gran acoplamiento* entre la aplicación y el DBMS: se debe hacer que la menor cantidad de partes posibles conozcan la interfaz del DBMS.

Para persistir estos objetos en la base de datos utilizamos:

Esquema de persistencia: es un conjunto reutilizable y generalmente expansible de clases que prestan servicios a los objetos persistentes. Debe traducir los objetos a registros y guardarlos en una BD y luego los traduce a objetos cuando los recupera de la BD.

La necesidad de usar una BD proviene de:

- la capacidad limitada de la memoria primaria (RAM): las BD se almacenan frecuentemente sobre almacenamiento secundario (discos rígidos), proveyendo maneras eficientes de acceder a los datos.

- almacenar objetos mayores que una ejecución de programa: que el objeto sobreviva a la ejecución que lo creó. Esta capacidad se llama ***persistencia***. La persistencia significa que los objetos se copian desde una memoria primaria rápida y volátil a una memoria secundaria, lenta y persistente.

Los datos persisten si sobreviven a la ejecución del proceso que los creó. Estos se almacenan en una BD o archivos que pueden ser leídos o modificados por otros procesos en un momento posterior. Los valores de todos los atributos del objeto, el estado general de éste y otra información complementaria se almacenan para su recuperación y uso futuro. En los ambientes OO, la idea de un objeto persistente extiende un poco más el concepto de persistencia.

DBMS: sistema de gestión de bases de datos que maneja el almacenamiento en la BD. El programador solo quiere una vista lógica de la BD y no decidir sobre cómo se hace el almacenamiento físico. Un DBMS debe proveer:

- *Concurrencia:* permite a múltiples usuarios trabajar con una BD común simultáneamente.
- *Recuperación:* si ocurre una falla de HW o SW, el DBMS debería volver la BD a un estado uniforme de datos.
- *Facilidad de consulta:* debería soportar una manera fácil de acceso a los datos en la BD.

Propiedades típicas que indican necesidad de un DBMS

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

Modelos de datos: los DBMS evolucionaron incluyendo modelos de datos, que son enfoques para realizar la persistencia de clases a BD en el contexto del diseño del SW. Determina la estructura lógica de una base de datos y de manera fundamental determina el modo de almacenar, organizar y manipular los datos

DBMSs Orientados a Objetos: su idea es almacenar a los objetos como tales, sin realizar ninguna unión lenta para conseguir el acceso a un objeto específico. La mayoría de los ODBMSs usan el lenguaje de programación directamente para almacenar y recobrar información, por lo que no se necesita una interfaz especifica con el DBMS, eliminando el problema de impedancia por completo. Esto también significa que el diseño de la BD se integra en el análisis y el diseño de la aplicación.

Enfoques	Características	Ventajas	Desventajas
Relacional	- Datos organizados en *tablas* compuestas por: * columnas: sus valores representan atributos. * filas: sus valores representan instancias particulares almacenadas de cada atributo.	- El lugar y la forma en que se almacenan los datos no tiene relevancia, por lo que es más *fácil* de entender por los usuarios. - Se usan desde hace mucho, son más maduros, altamente populares (dominante) y tienen un *soporte extensivo*	Problema de *impedancia*.

Jerárquico	- Datos organizados en una *estructura de árbol:* * cada registro tiene un único elemento o raíz. * los registros del mismo nivel se organizan de un modo específico que se usa de orden físico para almacenar la BD.	Especialmente útil para representar grandes volúmenes de información y datos muy compartidos.	Incapacidad de representar la *redundancia*.
OO	- Define una BD como una *colección de objetos* o elementos de SW reutilizables con métodos y funciones relacionados.	*Mejor modelo* conocido de BD: incorpora tablas, pero *no se limita* a ellas.	Bastante *reciente* comparado al relacional, que se usó mucho tiempo y posee más soporte.
De red	- La diferencia con el jerárquico radica en la modificación del concepto de nodo: *un nodo puede tener varios padres.*	Ofrece una *solución* ante el problema de *redundancia* de datos presente en el jerárquico.	Gran *dificultad para administrar los datos*, por lo que es mayormente utilizada por programadores y no usuarios finales.

Características de las BDOO:

1. soportar *objetos complejos*.
2. cada objeto debe tener una *identidad* independientemente de sus valores internos.
3. soportar *encapsulamiento* de datos y comportamiento de objetos.
4. soportar un mecanismo de estructuración de *tipos o clases*.
5. soportar la noción de *herencia* (jerarquía).
6. soportar sobreescritura y *ligadura tardía*. Relaciones entre objetos que se resuelven en tiempo de ejecución.
7. *completitud*: el lenguaje de manipulación debería poder expresar cada función calculable.
8. *extensibilidad*: poder agregar nuevos tipos.

Ventajas:

- Los objetos se almacenan en la BD como tales.
- No se necesita conversión del tipo de DBMS: las clases definidas por el usuario se usan como tipos en el DBMS.
- El lenguaje del DBMS puede integrarse con un lenguaje de programación OO.

Características o criterios que debería soportar una BD OO:

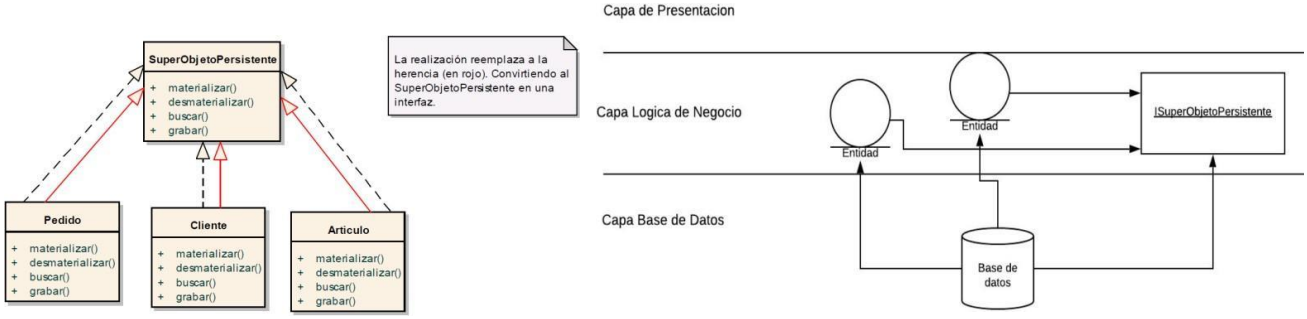
★ Debería poder almacenar objetos completamente (estado, identidad y comportamiento): no se necesitarían JOINS, sino que se accede directamente desde el objeto. Así también estaría incluido el comportamiento, permitiendo traer un objeto y utilizarlo sin necesidad de deserialización.

★ Las operaciones para manipular la BDOO deberían ser similares a las de los lenguajes de programación.

★ Las entidades almacenadas en la BDOO deberían estar relacionadas de la misma forma que los objetos nativamente, es decir, para navegar una BDOO se deberían soportar las mismas relaciones que para navegar objetos en memoria (agregación, composición, herencia y realización). Esta diferencia es clave, ya que en una BDR todos los objetos de una clase estarían en una misma tabla y se los podría recorrer secuencialmente, mientras que en una BDOO 2 objetos de la misma clase podrían no tener ninguna relación, y varios objetos de clases diferentes podrían relacionarse por composición con otro objeto que los contenga.

Correspondencia (mapping): entre clases y tablas, entre atributos y campos. Es decir, correspondencia de esquemas.

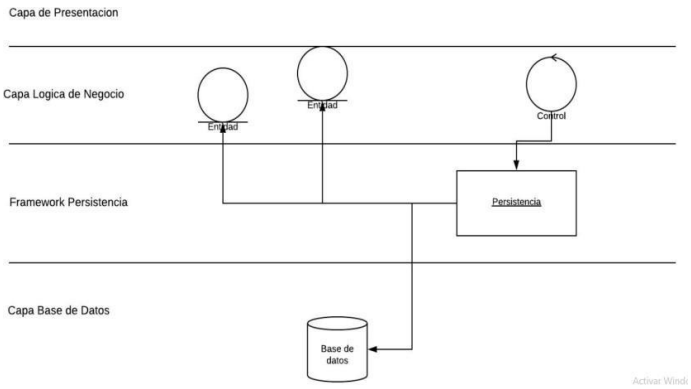
- *Esquema de Correspondencia Directa:*



Propone crear una clase de fabricación pura que se llame SuperObjetoPersistente, y en esa clase definir el método materialización, desmaterialización, junto con cualquier método que necesitemos para resolver la persistencia.
Entonces, todas las clases que necesitan 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 BD.
Desde el punto de vista arquitectónico el SuperObjetoPersistente se encuentra en la capa LN y cada clase persistente sabe cómo guardarse y recuperarse de la BDR.
Con esta solución se rompe el principio de diseño de:

- Genero fuerte acoplamiento entre mi capa de LN y mi capa de BD
- SRP, cohesión de las clases de dominio ya no tienen una responsabilidad, ahora se encargan ellas de saber como materializarse y desmaterializarse. También es responsable de la persistencia
- OCP, la clase se abre para agregar los métodos materializar y desmaterializar, también agregando el ID de la tabla en la estructura de la clase.

- *Correspondencia Indirecta:*



Correspondencia indirecta: Es una decisión arquitectónica. Agrego una capa de persistencia entre la LN y la BD, compuesto por clases de fabricación pura, que tienen el rol de establecer vínculos entre la capa de LN 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.

Autor: Andermatten – Martinez – Sánchez Díaz

El esquema de persistencia tiene un conjunto de clases o interfaces que definen los métodos y para integrar el esquema de persistencia a mi modelo de dominio, tengo que crear una clase de fabricación pura por cada clase del dominio que necesite persistencia.
Este esquema tiene 2 niveles, el primero son las interfaces que definen los lineamientos y después una clase de persistencia de BD para cada clase del dominio que necesite ser persistente. Se encarga de hacer el mapeo.
En las interfaces o clases abstractas está la decisión arquitectónica de cómo manejar la materialización, desmaterialización, identificación de un objeto con un ID, etc. Cada clase concreta tiene que implementar el comportamiento de acuerdo a los atributos que tiene la clase abstracta.
Para materializar y desmaterializar se debe guardar en la misma fila, manteniendo la referencia entre el puntero del objeto producto y el puntero ID de la fila en la tabla donde ese objeto estaba desmaterializado.

En este esquema se aplica el Principio de Hollywood (Inversión de dependencia), xq si no lo hiciéramos, las clase de negocio tendrán la responsabilidad de invocar al esquema de persistencia, entonces seguirán igual de acopladas. Para que esto no pase aplicamos el principio DIP donde una clase de control dentro del esquema invoca la persistencia, y cuando esto pasa el esquema de persistencia toma el control, donde busca el objeto que hay que hacer persistente y lo hace persistente sobre la base de datos, no nos llames, nosotros te llamamos, de esta forma mantengo la cohesión de las clases de dominio xq no las tengo que tocar, la persistencia se encarga de eso. Tiene bajo acoplamiento
Hay una dependencia entre la persistencia y la BD, si la cambio debo modificar el esquema de persistencia.

Desventaja: hace crecer mucho la estructura y es más lento que la otra solución.

Conversor de BD: es el responsable de la materialización y desmaterialización de los objetos.

Materialización: transformar registros (filas de tablas) de la BD en objetos en memoria.

- inmediata: recupera los objetos al instante.
- perezosa: difiere la materialización y recupera los objetos bajo demanda, cuando es absolutamente necesario, por cuestiones de rendimiento.

Desmaterialización: proceso inverso, transformar objetos en registros en la BD.

PATRONES DE PERSISTENCIA:

- *Patrón Identificador de Objetos:* propone asignar un identificador de objeto a cada registro y objeto. Es único, vincula objetos con registros evitando duplicados en una tabla sin equivocación. Es una forma consistente de relacionar objetos con registros y asegurar que la materialización repetida no de cómo resultado objetos duplicados.
- *Patrón fachada:* permite el acceso al servicio de persistencia mediante fachada, proporcionando una interfaz uniforme a un subsistema. La fachada no hace el trabajo, solo lo delega en objetos del subsistema. Se necesita una operación para recuperar un objeto dado un OID. También se necesita conocer el tipo del objeto que se va a materializar.
- *Patrón Conversor (Mapper) o Intermediario (Broker):* propone crear una clase responsable de materializar y desmaterializar los objetos. Agrega un intermediario por cada clase de dominio (genera el doble de clases). Utiliza objetos para establecer la correspondencia con los objetos persistentes. Enfoque de correspondencia indirecta.
- *Materialización con el método plantilla:* con el patrón TemplateMethod 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 Gestión de Caché:* es conveniente mantener los objetos materializados almacenados en una caché local para mejorar el rendimiento de y dar soporta a las operaciones de gestión de transacciones, con su 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 sobre la BD, más bien debe ejecutar una operación commit explícita. La respuesta a una operación depende del estado de la transacción del objeto (estado de los objetos para saber si se modificaron en función de su estado almacenado).
- *Diseño de una transacción con Patrón Command:* una transacción es una unidad de trabajo (conjunto de tareas) cuya terminación es atómica (se realizan todas o ninguna). En los servicios de persistencia se incluyen la inserción, actualización y eliminación de objetos, las cuales necesitan que se ordenen justo antes de la ejecución. La idea clave es representar cada tarea de la transacción como un objeto con un método ejecutar () polimórfico. Con el patrón se busca crear un objeto por cada operación a realizar en la transacción, haciendo una cola de operaciones (cola de objetos) permitiendo hacer un rollback (deshacer los cambios), si es necesario.
- *Materialización perezosa mediante un Proxy Virtual:* es un proxy para otro objeto (al que se quiere acceder) que materializa el sujeto real cuando se referencia por primera vez. Es una referencia inteligente. 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.

DISEÑO DE INTERACCIÓN HUMANO-MÁQUINA

Interacción Humano Computadora (IHC): disciplina relacionada con el diseño, evaluación e implementación de sistemas computacionales interactivos para uso humano, y el estudio de los principales fenómenos que los rodean.

Aspectos de la IHC a considerar:

- *uso y contexto:* problemas de adaptación a los computadores, su utilización y el contexto social de su uso.
  - trabajo y organización social: interacción social en el trabajo y modelos de actividad humana
  - áreas de aplicación: características de los dominios de aplicación
  - compatibilidad y adaptación hombre-computador: mejora la compatibilidad entre el objeto diseñado y su uso, adaptación de los sistemas a los usuarios ('customization') y viceversa (entrenamiento, facilidad de aprendizaje), guías al usuario (ayudas, documentaciones, manejo de errores).
- *características humanas:* comprensión de los seres humanos como sistemas de procesamiento de información, formas de comunicación entre humanos, requerimientos físicos y psicológicos.
  - procesamiento humano de la información: características del hombre como procesador de información (memoria, percepción, atención, resolución de problemas, aprendizaje y adquisición de experiencia, motivación)
  - lenguajes, comunicación e interacción: sintaxis, semántica, pragmática, interacción conversacional, lenguajes especializados.
  - ergonomía: características antropométricas y fisiológicas, relación con los ambientes de trabajo, disposición de pantallas y controles, limitaciones sensoriales y cognitivas, efectos de la tecnología, fatiga y salud, amueblamiento e iluminación, diseño de ambientes, diseño para usuarios con disminuciones físicas.

Patrones de conducta:

- *exploración segura:* "Déjame explorar sin perderme o meterme en problemas"
- *gratificación instantánea:* "Quiero lograr algo ahora, no luego"
- *cambios de parecer en la mitad del camino:* "Cambié de opinión sobre lo que estaba haciendo"
- *decisiones diferidas:* "No quiero responder a eso justo ahora, déjame terminar"
- *memoria espacial:* "El botón estaba aquí hace dos minutos, a dónde se fue?"
- *sólo teclado:* "Por favor no me hagas usar el mouse"
- *satisfacción:* "Es suficientemente bueno. No quiero invertir más tiempo aprendiendo a hacerlo mejor"
- *construcción incremental:* "Déjame cambiar esto. No quedó bien, déjame cambiarlo de nuevo. Así está mejor."
- *habituación:* "Esto funciona bien en otros lados, por qué no funciona aquí también?"
- *micro cortes (micro recreos):* "Estoy esperando el colectivo, déjame hacer algo útil por 2 minutos"
- *memoria prospectiva:* "Voy a dejar esto acá para recordarme a mi mismo ocuparme de esto más tarde"
- *repetición optimizada:* "Tengo que repetir esto ¿Cuántas veces?
- *consejo de otras personas:* "¿Qué dicen los demás sobre esto?
- *recomendaciones personales:* "Mi amigo me dijo que lea esto, así que debe ser bastante bueno"

Reglas de oro de Schneiderman: para el diseño de las interfaces de usuario **(CoARDEDeCoM)**

1. buscar siempre la **coherencia:** uso de iconos familiares, colores, jerarquía, utilizando el conocimiento previo del usuario, sin que tengan que aprender algo nuevo, para que puedan realizar lo que desean más rápidamente.
2. permitir el uso de short cuts (**atajos**): mientras el usuario adquiere experiencia, puede navegar y usar la interfaz más rápido y sin esfuerzo.
3. dar **retroalimentación** de información: el usuario debe saber dónde está y qué está pasando todo el tiempo, por

Autor: Andermatten – Martinez – Sánchez Díaz

lo que para cada acción, debe haber una respuesta del sistema para mantenerlo informado.

4. diseñar **diálogos** que tengan un fin: no dejar que el usuario adivine, debe saber el resultado de sus acciones.
5. permitir el manejo simple de **errores**: que reciba una solución simple y paso a paso para resolverlo rápido.
6. permitir **deshacer** las acciones con facilidad: regresar sobre sus propios pasos, retroceder o revertir sus acciones.
7. fomenta la sensación de **control**: que el usuario sea el que inicia las cosas, y el sistema funciona como desea.
8. reducir la carga de la **memoria** inmediata: interfaz lo más sencilla posible y con una jerarquía de información evidente. *Reconocimiento en vez de recuerdo*.

Heurísticas de Nielsen:

1) visibilidad del estado del sistema: mantener informado al usuario de lo que está ocurriendo.
2) relación entre el sistema y el mundo real: hablar el lenguaje de los usuarios mediante conceptos familiares haciendo que la información aparezca en un orden natural y lógico.
3) libertad y control del usuario: necesita una "salida de emergencia" claramente marcada para dejar el estado no deseado al que accedió, sin pasar por una serie de pasos (funciones de deshacer y rehacer).
4) consistencia y estándares: seguir las convenciones establecidas para que el usuario no se confunda.
5) prevención de errores
6) reconocer antes que recordar: hacer visibles los objetos, acciones y opciones, y las instrucciones a mano.
7) flexibilidad y eficiencia en el uso: interacción más rápida para usuarios expertos.
8) diseño estético y minimalista: los diálogos no deben contener información irrelevante o poco usada.
9) ayuda a los usuarios a reconocer, diagnosticar y recuperarse de los errores: mensajes de error en un lenguaje claro y simple, indicando precisamente el problema y sugiriendo una solución constructiva al problema.
10) ayuda y documentación: fácil de buscar, enfocada en las tareas del usuario, con una lista concreta de pasos a desarrollar y no muy extensa.

Estilos de interacción de usuario:

- *Lenguaje de comandos:* consiste en dar órdenes directamente al ordenador a través de un comando especial y los parámetros asociados para indicar al sistema qué hacer. Es la típica consola.
  - ✔ *Ventaja:* poderoso y flexible
  - χ *Desventaja:* difícil de aprender, pobre manejo de errores
- *Selección de Menús:* el usuario selecciona un comando de un conjunto de opciones visualizadas en pantalla que llevan a la ejecución de una acción asociada. Suele estructurarse jerárquicamente.
  - ✔ *Ventaja:* evita errores del usuario, requiere poco tipeo, fácil de recordar y visualización rápida
  - χ *Desventaja:* lento para usuarios experimentados, puede ser complejo si hay muchas opciones
- *Llenado de Formularios:* el usuario rellena los campos, la interacción es dirigida por el asistente personal o agente que colabora con el usuario.
  - ✔ *Ventaja:* ingreso de datos simple, reduce el esfuerzo del usuario.
  - χ *Desventaja:* ocupa mucho espacio en la pantalla, causa problemas cuando la opción del usuario no coincide con los campos del formulario
- *Lenguaje Natural:* el usuario emite un comando en lenguaje natural, con el uso de la voz o movimiento. Conocimiento de parte del SW del lenguaje del usuario y los movimientos exactos.
  - ✔ *Ventaja:* accesible a usuarios casuales y fácilmente extensible
  - χ *Desventaja:* requiere más tipeo
- *Manipulación Directa:* el usuario interactúa directamente con los objetos de la pantalla, gracias a las pantallas gráficas de alta resolución y los dispositivos apuntadores.
  - ✔ *Ventaja:* interacción rápida e intuitiva, fácil de aprender
  - χ *Desventaja:* difícil de implementar, adecuado sólo cuando hay una metáfora visual para tareas y objetos

Diseño de interfaz de usuario: consiste en el diseño de herramientas de HW y SW enfocado en la experiencia de usuario y la interacción. Normalmente es una actividad multidisciplinar que involucra varias ramas del diseño y el conocimiento (diseño gráfico, industrial, web, de SW y ergonomía). Busca que estas herramientas sean más atractivas y familiares para los usuarios. Diseñar centrado en el usuario es importante debido a que si no se siente cómodo con las herramientas, no las utilizará, y su elaboración no habrá tenido sentido alguno.

Evaluación de la interfaz de usuario: es el proceso de evaluar la forma en que se utiliza una interfaz y verificar que cumple con los requerimientos del usuario. Se lleva a cabo contra una especificación de la usabilidad para evaluar su nivel de adecuación basada en ***los atributos de usabilidad***:

- aprendizaje: ¿cuánto tiempo tarda un usuario nuevo en ser productivo con el sistema?
- velocidad de funcionamiento: ¿cómo responde el sistema a las operaciones de trabajo del usuario?
- robustez: ¿qué tolerancia tiene a los errores del usuario?
- recuperación ¿cómo se recupera de los errores del usuario?
- adaptación ¿está muy atado a un único modelo de trabajo?

Usabilidad: grado de facilidad en el uso del sistema interactivo. La usabilidad puede ser definida en el contexto de la aceptabilidad de un sistema

- decrementa los costos: previene cambios en el SW antes de su uso y elimina parte del entrenamiento necesario.
- incrementa la productividad: menor tiempo para realizar las tareas y de aprendizaje y menos errores.
- no todas las características de usabilidad tienen el mismo peso en un diseño.

❖ *inconvenientes:*

- brecha de evaluación: inconvenientes en la evaluación y/o interpretación de la presentación por factores ergonómicos (texto difícil de leer, información importante presentada con poco contraste, ítems agrupados de forma inadecuada), falta de feedback.
- brecha de ejecución: inconvenientes en la elaboración del plan de acción del usuario para lograr su tarea, por desconocimiento del usuario de los efectos que producen las posibles acciones, feedback inadecuado o inexistente de las acciones del usuario, cambios en la forma de operar un comando en versiones nuevas.

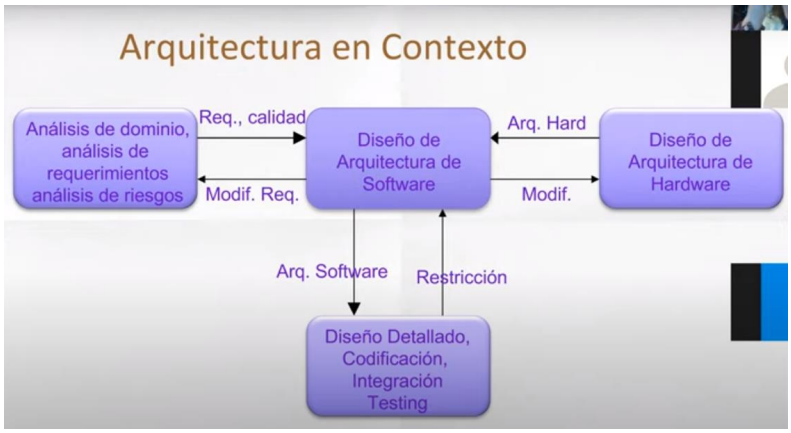
Idiomas: tipos o estilos de interfaces reconocibles, cada uno con su vocabulario de los objetos, acciones y efectos visuales que contiene. Las aplicaciones fáciles de usar están diseñadas para ser familiares. (forms, editores de texto y gráficos, calendarios, navegadores, hojas de cálculo, reproductores, juegos, páginas web, social, e-commerce).

Patrones de comportamiento para el diseño de interfaces

- navegación
- disposición de elementos
- acciones y comandos
- estructura física
- formularios y controles

Unidad Nro. 3: Diseño de Arquitecturas de Software

DISEÑO ARQUITECTÓNICO



¿Qué es el diseño arquitectónico? Lo más importante y costoso son los requerimientos, en segundo lugar viene la arquitectura.

Es lo primero que se diseña, nosotros diseñamos arquitectura de software, no hardware, entonces acá decidimos qué hardware vamos a necesitar para poder desplegar el software que hemos diseñado y construido.

La arquitectura es tomar los requerimientos esenciales y asignarlos a una tecnología específica, es el plan del diseño(Su primer etapa), es decir si hacemos diseño arquitectónico estamos haciendo un diseño general, estratégico (que no entra mucho en detalle), Nos hacemos la pregunta de como hago para resolver los requerimientos que he identificado en etapas anteriores. Los requerimientos que le queremos dar solución es a los requerimientos no funcionales.

La arquitectura es como un cimiento para el producto, captura la estructura del sistema en términos de componentes y cómo estos interactúan. Entiende cómo debe organizarse un sistema y cómo diseñar su estructura global

Rol del arquitecto

Es un rol importante, no suele ser una persona sino un equipo. Responsable de definir el documento de la arquitectura(Documento principal), lineamiento de diseño, lineamiento que el equipo va a implementar, prototipo arquitectónico (válida si ciertas decisiones arquitectónicas funcionan o no). Comunicarse y negociar con muchos involucrados

- revisar y negociar los requerimientos
- documentar la arquitectura
- comunicar la arquitectura, asegurando que los interesados la comprendan
- direccionar los RNF a la arquitectura
- configurar la arquitectura de HW
- trabajar con el administrador del proyecto, ayudando en la planificación, estimación, distribución de las tareas y la calendarización del proyecto
- se asegura que la arquitectura sea respetada

Ventajas de diseñar y documentar la arquitectura

- *Comunicación con los participantes:* la arquitectura es una presentación que sirve para usarse como enfoque para la discusión de un amplio número de participantes.
- *Análisis del sistema:* aclara la arquitectura del sistema y determina si se cumplen los requerimientos críticos.
- *Reutilización a gran escala:* es posible desarrollar arquitecturas de línea de productos donde la misma se reutilice mediante una amplia gama de sistemas relacionados.

Conflictos arquitectónicos

- Utilizar componentes de granularidad alta mejora la performance pero reduce la mantenibilidad
- La introducción de datos redundantes mejora la disponibilidad pero hace la seguridad mas dificil.
- La localización de aspectos de seguridad relacionados usualmente significa mas comunicacion por lo tanto degrada la performance

Modelado de la Arquitectura: El modelo arquitectónico mapea los RF del análisis a una arquitectura tecnológica, y también trata con los RNF. No existe la solución "perfecta", sino que se debe escoger la óptima para el conjunto de circunstancias existentes. Considerar información sobre:

- volúmenes de datos
- funcionalidad demandada en el negocio
- distribución geográfica
- distribución del procesamiento de datos
- dónde se guardarán los datos
- cuáles procesos se ejecutarán en qué procesadores y que tanta comunicación se requerirá entre ellos

Salidas del modelo arquitectónico: *la salida del proceso de diseño arquitectónico consiste en un modelo que describe la forma en que se organiza el sistema como un conjunto de componentes comunicados entre sí.*

- distribución geográfica de los requerimientos de computación
- componentes de HW: máquinas y servidores
- configuración y cantidad de niveles de HW
- mecanismos y lenguajes de comunicación de la red
- SOP
- paradigma de desarrollo
- frontend y backend
- sistema de administración de bd
- ubicaciones de los procesos y de los datos físicos
- estrategias de sincronización de los datos distribuidos físicamente

Actividades del proceso de arquitectura de software: (Descripción del proceso)

1) *Determinar los Requerimientos arquitectónicos*: Creación de un modelo de requerimientos que conducirán al diseño arquitectónico y su priorización.

* Entrada: RF y RNF de los involucrados (stakeholders) + atributos de calidad
* Salida: Documento con los requerimientos arquitectónicos

Al conjunto de requerimientos definidos (Funcionales y no funcionales) los vamos a analizar para ver cuales son significativos y cuales me condicionan la arquitectura. Si estos requerimientos me hacen tomar una decisión arquitectónica, los separamos y los clasifico. Se resuelve estos requerimientos desde la arquitectura (pasar a la siguiente actividad)
Si los requerimientos me van a hacer tomar una decisión arquitectónica los voy a Priorizar:

* **Alto**: conducen la arquitectura (Tiene que estar listo desde la primera iteración)
* **Medio**: necesitará ser soportado en alguna etapa, pero que pueden ser desarrollados en próximas iteraciones (Disponible desde la 2da o 3era o 4ta iteración)
* **Bajo**: son parte de la lista de deseos. Las soluciones que los incluyen son deseables pero no conducen el diseño. (Últimas iteraciones)

Entonces el criterio de priorización puede cambiar según el tipo de producto, el ciclo de vida del proceso desarrollo.

EL PUD dice que empecemos por la funcionalidad (Centrado en la arquitectura) que es lo significativo por la arquitectura primero y después las demás.

2) *Proceso de Diseño Arquitectónico*: Definir la estructura y las responsabilidades de los componentes.

* Entrada: Requerimientos significativos para la arquitectura
* Salida: Modelo arquitectónico (Documento de arquitectura y vistas arquitectónicas)

La salida de este proceso es el modelo arquitectónico (Documento de arquitectura y vistas arquitectónicas)
Esta actividad se divide en dos tareas.

Elegir un framework:
El framework es un conjunto de patrones que vamos a utilizar para estructurar nuestro producto, son patrones de alto nivel (Nivel de capas o componentes). Los patrones se aplican (significa que hay que construir los componentes que implementa estos patrones). Un patrón es una plantilla genérica. (Patrones arquitectónicos, los cuales nos servirán para resolver requerimientos)

Diseñar y distribuir componentes:
se deben identificar los componentes principales de la aplicación, los servicios o interfaces que soportan y proveen, sus responsabilidades y las dependencias con otros. También se deben identificar particiones en la arquitectura que son candidatas para distribución entre servidores de red.

Es la construcción de componentes y el acoplamiento, esta tarea da origen a las vistas arquitectónicas (Modelan la arquitectura) y la documentación sobre las decisiones que hemos tomado en la arquitectura.

3) *Validación*: implica "probar" la arquitectura contra los requerimientos(SPA). Comprobar que el documento de la arquitectura cumple con los requerimientos especificados, si los cumple la validación está bien, si no los cumple retrocedemos hasta donde haga falta para lograr que se cumpla.

Ayuda a incrementar la confianza del equipo de diseño en que la arquitectura cumpla su propósito. Debe ser realizada por un arquitecto distinto a quien diseñó la arquitectura, con las restricciones de tiempo y presupuesto del proyecto

* Entrada: vistas y documento de la arquitectura
* Salida: vistas y documento de la arquitectura validados

Hay dos técnicas principales para la validación: buscan identificar potenciales defectos, fallas, debilidades y áreas de riesgo en el diseño para que puedan ser mejoradas antes de la implementación y construcción.

- *Escenarios:* testeo manual de la arquitectura usando casos de prueba. Son artefactos simples. Implican la definición de estímulos que tendrán impacto en la arquitectura y se trabaja la respuesta de ella.
- *Prototipos:* crean un arquetipo simple de la aplicación deseada. Son versiones mínimas, restringidas de la aplicación creados específicamente para probar algún aspecto riguroso o pobremente comprendido en el

diseño. Tienen la capacidad de evaluar los requerimientos en forma detallada Son utilizados para dos propósitos: en ambos casos proveen una evidencia que de otra forma no se puede validar.
* prueba de conceptos: ¿puede la arquitectura diseñada ser construida satisfaciendo los requerimientos?
* prueba de tecnología: ¿la tecnología elegida se comporta como es esperado?

REQUERIMIENTOS NO FUNCIONALES:

¿Qué son? Los RNF definen cualidades o atributos que deben estar presentes en el producto de SW resultante. Juegan un papel crucial en el diseño y desarrollo del sistema de información. Pueden definirse como *consideraciones o restricciones asociadas a un servicio del sistema*. Suelen llamarse también requerimientos de calidad o no comportamentales en contraste con los comportamentales. Pueden ser críticos con los funcionales.

Dificultades asociadas a los RNF a la hora de diseñar SW:

- χ No hay reglas para determinar cuándo se obtuvo una solución óptima.
- χ Tiene buenas y malas soluciones, NO correctas e incorrectas
- χ Los problemas relacionados con los RNF pueden ser síntomas de problemas mayores
- χ Deben ser objetivos y testeables

Impacto en el Modelado de la Arquitectura

Los RNF se utilizan para determinar los Requerimientos Arquitectónicos. Si un RNF es crítico, puede determinar la elección del Modelo Arquitectónico a diseñar, debido a la estrecha relación entre los RNF y la Arquitectura de SW, sacrificando a otros RNF de prioridad alta.

Modelos Arquitectónicos según RNF (cómo sugiere cada uno diseñar la arquitectura si son requerimientos críticos) y relación con los atributos de calidad del producto:

- *Desempeño (Performance):* localiza operaciones críticas en un número reducido de subsistemas con la menor comunicación posible entre ellos, es decir, componentes de granularidad gruesa para reducir componentes de comunicación. Analiza la cantidad de recursos utilizados bajo determinadas condiciones establecidas (tiempo de procesamiento, de respuesta, rendimiento, capacidad, concurrencia de usuarios, almacenamiento, tamaño). Son SPA y de prioridad ALTA. ***Impacto arquitectónico: se mejora la performance, se reduce la comunicación, pero se reduce la modularidad/mantenibilidad ya que se aumenta la granularidad, y se sacrifica la cohesión.***

Nombre	Descripción	SPA-prioridad	Justificación
Concurrencia	-Es necesario controlar la concurrencia para cada puesto. -El sistema debe soportar la ejecución concurrente de tales aparatos durante tanto tiempo. -El producto debe dar soporte hasta tantos usuarios sin afectar la performance de la aplicación.	Sí. Alta.	-La BD debe soportar el mecanismo de concurrencia para soportar las múltiples conexiones de acceso y actualizaciones de los datos. -El módulo de SW que resuelva la comunicación con los aparatos debe asegurar la concurrencia.
Tiempo de respuesta	-El tiempo de respuesta no deberá superar tantos segundos para tal tarea.	Sí. Media/Baja	El módulo de SW que haga esa tarea debe asegurar esta performance.

- *Seguridad:* utiliza una estructura en capas con los recursos más críticos protegidos en capas internas y un alto nivel de validación aplicado a las mismas. Protege información y datos apropiados a sus tipos y niveles de autorización. Asegura que los datos son accesibles sólo a aquellos autorizados. ***Impacto en la arquitectura: se***

divide la arquitectura en capas, agregando capas de seguridad y validaciones, lo que significa más comunicación, por lo tanto degrada la performance.

- *Protección:* incluye operaciones relacionadas con la protección en un número reducido de subsistemas. Reduce costos y problemas de validación.
- *Disponibilidad:* incluye componentes redundantes de forma que puedan reemplazarse y actualizarse sin detener el sistema. ***Impacto en arquitectura: la introducción de nuevos componentes (costos de Redundancia) mejora la disponibilidad, pero dificulta la Seguridad.***
- *Mantenibilidad:* utiliza componentes auto-contenidos de granularidad fina que puedan cambiarse con facilidad. Los productores de datos separados de los consumidores y las estructuras de datos compartidas deben evitarse. Instalación de actualizaciones y nuevas versiones. Compuesto por componentes discretos de manera que el cambio en uno de sus componentes tiene impacto mínimo en el resto. Permite reusabilidad. Capacidad de probar y evaluar el impacto o diagnosticar un producto. Ser modificado sin introducir defectos o degradar la calidad del producto. ***Impacto en la arquitectura: alta cohesión pero problemas de performance.***

Otros impactos (atributos ISO9126):

- *Compatibilidad:* intercambiar información con otros productos, sistemas o componentes.

Nombre	Descripción	SPA-prioridad	Justificación
Conexión con otros dispositivos o máquinas	Se debe plantear una interfaz que permita el intercambio de información bidireccional/unidireccional con el dispositivo o máquina.	Sí. Alta.	Módulo (servicio) que resuelva la comunicación con el dispositivo o maquina y sea intermediario con la BD.
Soporte WEB	El producto proporcionará funcionalidad WEB para tal tarea.	Sí. Media/Alta.	Servidor WEB, lenguaje de programación para resolver interfaz WEB.
Comunicación con servicio Posnet	Comunicación con HW de autorización de transacciones con tarjeta de crédito/débito.	No	La comunicación con la ATC para la autorización de transacciones es resuelta por el SW del posnet, por lo que no afecta la arquitectura.

- *Usabilidad:* puede ser utilizado por usuarios específicos para cumplir con sus objetivos, ser apropiado para sus necesidades, impresión inicial, documentación asociada al producto o sistema. Fácil de operar y controlar. Protege los usuarios de cometer errores. Interacción placentera y satisfactoria a los usuarios. Suelen no ser SPA.

Nombre	Descripción	SPA-prioridad	Justificación
Logo	Se debe incluir el logo en todas las pantallas del sistema, en la misma posición y tamaño.	No.	La inclusión de una imagen en una pantalla no afecta la arquitectura del SW, está resuelto por el lenguaje de programación utilizado.

- *Confiabilidad:* está operativo y es accesible cuando es requerido. Tiempo durante el cual el sistema está en condiciones de ser utilizado. Tolerancia a fallas y la capacidad de recuperación, restaurar la información afectada y restablecer el estado deseado del sistema.
- *Portabilidad:* puede ser transferido de un HW, SW u otro entorno operativo a otro o adaptado, capaz de ser reemplazado. Permite escalabilidad, hacer crecer el producto. Facilidad de instalación.

Clasificación (o áreas que deben resolver):

- *Restricciones técnicas:* restringen opciones de diseño *por razones técnicas*, para especificar ciertas tecnologías que la aplicación debe usar. Ejemplo: "solo tenemos diseñadores Java", "la base de datos debe correr con Windows XP". Usualmente no son negociables.
  - de interoperatividad
  - de implementación
- *Restricciones de negocio:* restringen las opciones de diseño *por razones de negocio*. Ejemplo: "las licencias son muy costosas, nos vamos a una versión de código abierto", "hay que desarrollar interfaz con el producto X que ya existe en la empresa". Usualmente no son negociables.
  - de estándares
  - de entrega
  - éticos
  - legales
- *Atributos de calidad del producto (ISO9126):* definen los requerimientos de una aplicación. Direccionan aspectos de interés para los usuarios y otros involucrados.
  - disponibilidad
  - performance: concurrencia, uso de recursos, tiempo de respuesta
  - usabilidad
  - seguridad: lógica y física
  - confiabilidad
  - portabilidad
  - interfaz: HW, SW, comunicación, usuario

PATRONES ARQUITECTÓNICOS

PATRONES: Existen patrones para distintos niveles de solución:

- De bajo nivel y detallados: idiomas (lenguajes de programación). Aplicación de un patrón en un lenguaje en particular.
- De nivel medio: PATRONES DE DISEÑO (objetos comunes y patrones de clases) -> Apunta a resolver la problemática de una clase, o dos o tres clases en particular. Como resolver el rol que esa clase tiene en el dominio de la solución, es más puntual la aplicación de un patrón de diseño.
- De alto nivel: PATRONES ARQUITECTÓNICOS (componentes y módulos) -> Tiene un nivel de abstracción más genérico, apuntan a analizar a niveles de capas, de subsistemas. Cómo voy a estructurar mi producto de software en términos más generales, para resolver problemas más generales.

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

Patrones vs. Estilos arquitectónicos: suele ser difícil distinguirlos. A veces se los ve como sinónimos.

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

Platónicos vs. Embebidos

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

1. Layered: arquitectura Estratificada o en capas: es la más común, muchos desarrolladores asumen que todos los sistemas deberían ser estratificados. Organiza el sistema en una pila de capas que agrupan módulos con funcionalidad coherente (relacionada). Cada capa actúa como máquina virtual de la capa de arriba. En un estilo estratificado simple, las capas sólo pueden usar la capa que está directamente debajo. La restricción significa que las capas subsiguientes más abajo están ocultas. Aplica a elementos de código, es parte del ***tipo de vista módulo***.

Es la forma en la que la mayoría de las aplicaciones se están estructurando hoy en día, nos ayudan a estructurar el código. Es estática (Estructura) pertenece a un tipo de vista estática (Vista de Módulo) y los demás que siguen son de ejecucion(runtime).
Nos muestra como vamos tomar todos los componentes de códigos, los vamos a dividir en capas y que componente de código le voy a asignar en cada capa.
Las capas solo se comunican con las capas adyacentes (Principio de diseño no hables con extraños, o solo habla con amigos)

Cohesión y acoplamiento a nivel de capa: Los componentes de código que estan dentro de una capa tienen que estar relacionados entre sí, para que tengan la menor cantidad de relación con los componentes de la otra capa -> Esto logra alta cohesión en la capa y bajo acoplamiento entre capas y debería existir acoplamiento 0(cero) entre capas que no son adyacentes

**resultado:* la restricción trata con los atributos de modificabilidad, portabilidad, reusabilidad. Dado que las capas dependen sólo de la inmediata inferior, las capas subsecuentes pueden ser intercambiadas. Las capas superiores dan más oportunidades de sustitución frente a un problema de performance.

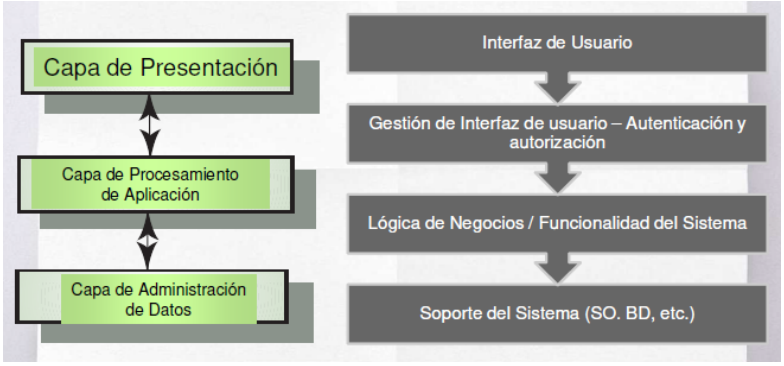
**variantes:* evitan la restricción, de modo que las capas puedan comunicarse con capas de más abajo. Otras variantes usan capas compartidas, donde cada capa puede usar estas capas verticales o compartidas. Este uso tensiona la definición de capas hasta el límite.

* *cuando se usa:* al construirse nuevas facilidades encima de los sistemas existentes, cuando el desarrollo se dispersa a través de varios equipos de trabajo, o cuando existe un requerimiento de capa multinivel.
* *ventajas:*

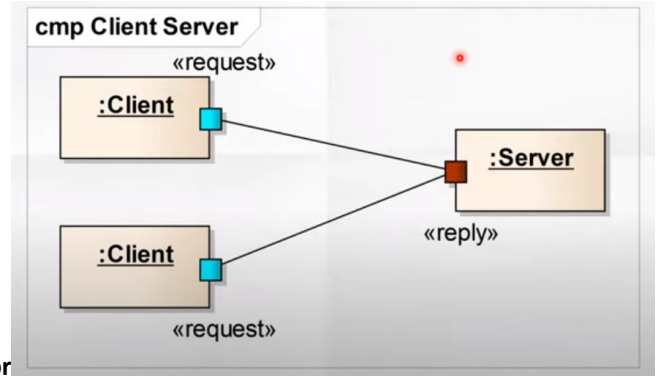
- sustitución de capas completas en tanto se conserve la interfaz.
- en cada capa pueden incluirse facilidades redundantes para aumentar la confiabilidad.

* *desventajas:*

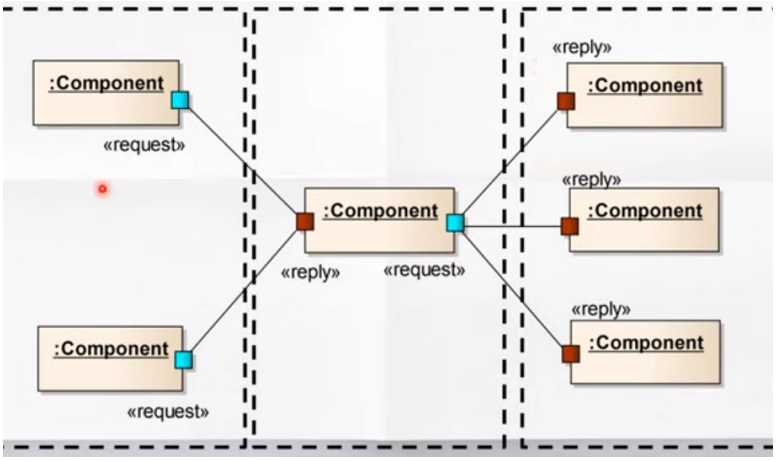
- difícil mantener la separación: capas de nivel superior se comunican con las de nivel inferior no adyacente.
- problemas de rendimiento por los múltiples niveles de interpretación de una solicitud de servicios mientras se procesa en cada capa.



2. Cliente Servidor N-Tiered: Modela un conjunto de componentes que funcionan como cliente (capa externa) o como servidor. La comunicación es síncrona, el cliente pide servicios a los servidores y no puede avanzar hasta que el servidor le responda. Es parte de la ***vista de distribución***.
Los clientes conocen los servicios y deben saber cómo buscarlos, pero los servidores no necesitan conocer a los clientes. La distribución es flexible, no hay restricciones para la distribución multi capas de la aplicación. Hay una separación de intereses, son capas diferentes y claramente divididas en presentación (clientes), servidor de lógica de negocio y servidor de administración de base de datos



Cliente servidor



Cliente Servidor & N-Tier

3. Arquitectura MVC: el sistema se estructura en tres componentes lógicos que interactúan entre sí para incrementar flexibilidad y reutilización:
- ***modelo:*** objeto aplicación, maneja los datos del sistema y las operaciones asociadas a esos datos. Frente a cambios, el modelo se encarga de avisar a las vistas que dependen de él
- ***vistas:*** objeto presentación en pantalla (interfaz de usuario), define y gestiona cómo se presentarán los datos al usuario (refleja el estado del modelo). La vista se actualiza a sí misma. Se pueden crear varias vistas de un modelo, para ofrecer diferentes presentaciones.
- ***controlador:*** dirige la interacción del usuario y pasa esas interacciones a vista y modelo (recibe peticiones de la vista sobre el modelo, interacciona con el modelo, devuelve el resultado a la vista).

Desacopla el modelo de las vistas estableciendo un modelo de suscripción/notificación. Separa presentación e interacción de los datos del sistema.

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

* *ventajas:*

- permite que los datos cambien independientemente de su representación y viceversa.

- soporta distintas representaciones de los mismos datos y los cambios en una representación se muestran en todos.

* *desventajas:*

- puede implicar código adicional y complejidad cuando el modelo de datos y las interacciones son simples.

4. Publish-Suscribe: es parte de la ***vista de ejecución (runtime)***. Componentes independientes publican eventos y otros se suscriben a ellos. Desacopla publicadores de suscriptores. Ambos se relacionan a través de ***tópicos***.

Componentes estáticos y vinculos(dinamico) es decir comunicación entre componentes
Frente a un cambio de estado(Algun valor en algun atributo se modifico; hay un evento que hace que cambie de estado) en un componente haya otro componentes que manifestaron interés en ese cambio de estado, se tienen que suscribir.

Nivel de acoplamiento bajo, ya que el publicante y suscriptor no se conocen (no se comunican) y si hay una comunicacion es atraves del trópico

Se pueden agregar y quitar publicante/suscriptor.

Pull y push, pull notifica que tiene que actualizar y push actualiza forzosamente

Cuando hablamos de filtrado por tópicos, decimos que los mensajes son publicados por “temas” o canales. Por ejemplo el clima -> el tópico es Información del clima

Mensajeria asincronica, no se pierde informacion.

Publicantes: ignoran el motivo global por el cual el evento es publicado. Pueden aparecer y desaparecer de un tópico dinámicamente sin afectar a ningún otro componente, lo que le da flexibilidad. Envían mensajes sin saber quién los va a leer.
Suscriptores: ignoran por qué o quién publica el evento y dependen sólo del evento no de quien lo publica. Pueden suscribirse y des-suscribirse dinámicamente a un tópico. Eligen tópicos que les interesa y leen cuando hay mensajes nuevos.

* *Propiedades:*

- mensajería muchos a muchos: los mensajes publicados son enviados a todos los subscriptores que están registrados en el tópico. Muchos publicantes pueden publicar en el mismo tópico, y muchos suscriptores pueden escuchar en el mismo tópico.
- calidad del servicio (QoS) configurable: además de mensajes seguros/no seguros, el mecanismo de comunicación

subyacentes puede ser punto-a-punto (un mensaje distinto para cada suscriptor) o multicast/broadcast (manda un mensaje que todo suscriptor recibe).
- bajo acoplamiento: los publicantes no saben quién recibe su mensaje y los suscriptores no saben quién lo envía. No hay vínculo directo entre ellos.

El bus de evento agrega una capa de indirección entre productores y consumidores, lo que podría dañar la performance del sistema. Propiedades de los canales (buses) de eventos:

- durables: garantizan que cualquier mensaje que aceptan no se perderá durante una falla. La durabilidad se logra escribiendo los eventos en un almacenamiento confiable.
- entrega en orden: garantizan entrega en orden o priorizada de eventos.
- entrega en lotes: los eventos se acumulan en lotes para evitar cúmulos de eventos similares.

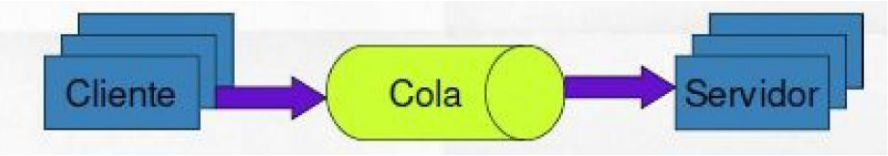
* *cuando se usa:* en arquitecturas muy flexibles y adecuadas para aplicaciones que requieren ***mensajes asíncronos***1...*, *...1 y *...* entre componentes, donde no se conocen ni los publicadores ni los suscriptores de antemano.

* *ventajas:*
- no hay vínculo directo entre publicantes y suscriptores. Los publicantes no saben quién recibe su mensaje, y los suscriptores no saben qué publicante lo envía.
- desacopla los componentes que producen los eventos de quienes los consumen, haciendo que la arquitectura sea mantenible y evolucionable.

* *desventajas:*
- el bus de eventos agrega una capa de indirección entre productores y consumidores, pudiendo afectar la performance del sistema.

5. Messaging: desacopla emisores de receptores almacenando los mensajes en una cola intermedia. El emisor envía un mensaje al receptor y sabe que eventualmente será entregado, aunque la red esté caída o el receptor no esté disponible (***comunicación diferida***).Los cambios en los formatos de los mensajes de los receptores pueden causar cambios en las implementaciones de los emisores. Los formatos de mensajes auto-descriptivos reducen esta dependencia. El cliente envía la petición a la cola y le delega a ésta el envío, continuando con sus actividades.

Un determinado componente de software quiere enviar peticiones a otro, entonces se acumulan en la cola



* *Propiedades:*

-***comunicación asincrónica:*** el cliente envía pedidos a una cola donde se almacenan hasta que una aplicación los remueve, y continúa sin esperar respuesta.

-QoS configurable: la cola es configurable para una entrega de mayor velocidad y no confiable o de menor velocidad y confiable. Las operaciones de la cola pueden coordinarse con transacciones de BD.

- bajo acoplamiento: el cliente es inconsciente de cuál servidor recibe el mensaje y el servidor es inconsciente de qué

cliente vino. No hay vínculo directo entre ellos.

* *cuando se usa:* provee una solución resistente para aplicaciones en las cuales la conectividad es transitoria, tanto debido a la red como a la poca confiabilidad de los servidores. Es apropiada cuando se deben enviar muchos mensajes y no se puede esperar a que llegue cada uno, o cuando el cliente no necesita respuesta inmediata. Se puede implementar para pedido-respuesta sincrónico utilizando un par de colas pedido-respuesta.

* *ventajas:*

- promueve el bajo acoplamiento, permitiendo alta modificabilidad, dado que emisores y receptores no se conectan.

* *desventajas:*

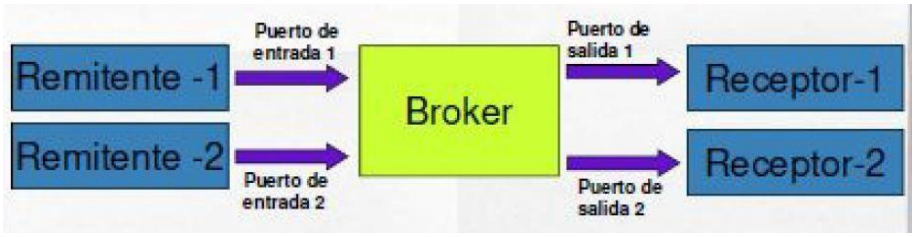
- el bus de eventos agrega una capa de indirección entre productores y consumidores, pudiendo afectar la performance del sistema.

8. Broker: expone puertos para diferentes tipos de entradas y salidas de mensajes. Luego los clientes se conectan a la entrada y salida para enviar mensajes. El broker es encargado no sólo de traducir dichos mensajes de un puerto a otro, sino también de enrutarlos según corresponda. También actúa como concentrador, ya que puede atender múltiples clientes para cada tipo de mensaje. ***Comunicación asíncrona***.

Dos funciones principales:
1- Formatea el mensaje (transformar el mensaje)
2- Enruta el mensaje formateado (Entregando al receptor en el formato que corresponde) (A quien se lo envía)
Motivación: Cuando hay un problema de compatibilidad de formato entre los componentes que producen la información y los componentes que reciben la información.

La información entra en un formato, el broker transforma ese formato, para que el receptor pueda interpretar y de ahí el componente receptor hace lo que tenga que hacer.

Hay que comunicar nuestro sistema a un sistema externo o componentes o dispositivos de hardware, o al revés. Entonces transformamos a un formato para que nuestro sistema o al revés puedan entender



* *Propiedades:*

- arquitectura Hub-and.spoke: el agente actúa como concentrador de mensajes y los remitentes y receptores se conectan al agente como rayos, por medio de puertos asociados a un formato específico de mensaje.
- realiza ruteo de mensajes: el agente incrusta la lógica de procesamiento para entregar un mensaje recibido en un puerto de entrada en el puerto de salida.

- realiza transformación de mensajes: el agente lógico transforma el tipo de mensaje de origen recibido en el puerto

de entrada en el tipo de mensaje de destino requerido por el puerto de salida.

* *cuando se usa:* son cuando los componentes intercambian mensajes que requieren grandes transformaciones entre los formatos de origen y destino. Por ejemplo, cuando tomamos mensajes de un sistema externo que utiliza un formato de mensaje incompatible con nuestro sistema.

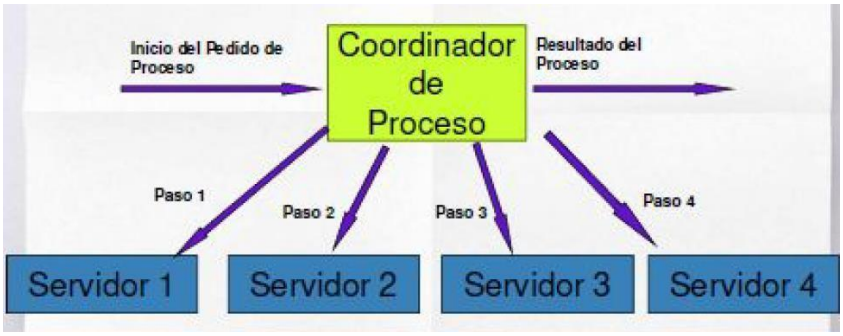
* *ventajas:*

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

9. Arquitectura Coordinador de Proceso (ProcessCoordinator): se utiliza para resolver peticiones muy complejas que pueden separarse en pasos. Se encapsulan las etapas de procesamiento en varios servidores distintos que resuelven cada uno una parte. Luego se incluye un coordinador que es quien sabe en qué orden y qué servidores resolverán la petición. Se reduce así la complejidad y se desacoplan los componentes, ya que los servidores son ignorantes de dónde se los utiliza y el coordinador no sabe cómo resuelve cada servidor su tarea, sólo que lo hace.

Se utiliza para procesos de negocios complejos. Un proceso de negocio complejo es quien tiene varios pasos, y también tiene una lógica de orden de ejecución de esos pasos, con una única salida.
Estructuramos cada uno de estos servidores para que resuelvan cada paso, y al coordinador que va a invocando los pasos en funcion a la logica que tiene definida. Cuando recibe la respuesta de las peticiones, avanza al siguiente paso hasta que termine el resultado del proceso.

Ejemplo práctico: otorgamiento de título, préstamo hipotecario.



* *Propiedades:*

- encapsulamiento de proceso: encapsula la secuencia de pasos necesarios para completar un proceso de negocio. El coordinador es uno único de definición para el proceso de negocio, haciendo más fácil de comprender y modificar. Recibe un requerimiento de inicio de proceso, llama a los servidores en el orden definido y emite los resultados.

- bajo acoplamiento: los componentes del servidor son inconscientes de su rol en el proceso de negocio general, y del orden de los pasos del proceso. Los servidores simplemente definen un conjunto de servicios que pueden ejecutar y el coordinador los llama cuando es necesario.

-***comunicaciones flexibles***: entre el coordinador y los servidores, pueden ser ***sincrónicas*** (el coordinador espera hasta que el servidor responda) o ***asincrónicas*** (el coordinador provee una llamada o respuesta cola/tópico, y espera hasta que el servidor responda).

* *cuando se usa:* es utilizado comúnmente para implementar procesos de negocio complejos que deben publicar requerimientos a algunos componentes servidores diferentes, ya que encapsula la lógica de negocio del proceso en un solo lugar, facilitando su manutención.

* *ventajas:*

- es más fácil cambiar, administrar y monitorear la performance del proceso encapsulando la lógica en un lugar.

VISTAS ARQUITECTÓNICAS

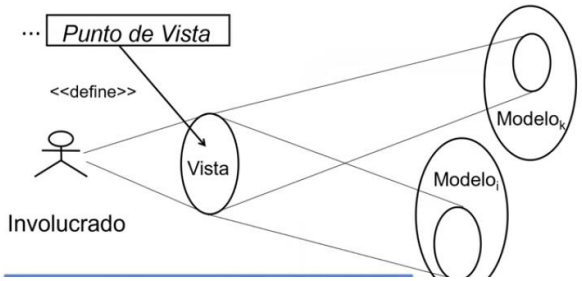
Arquitectura: conjunto de decisiones significativas que tomamos sobre el producto para cumplir con los requerimientos. Explica cómo se satisfacen los RF y RNF. Es el artefacto más importante para manejar los diferentes puntos de vista que pueden tener los usuarios en los diversos momentos a lo largo de la vida del proyecto y controlar el desarrollo iterativo e incremental de un sistema. La arquitectura tiene que ver con la estructura, el comportamiento, el uso, la funcionalidad, el rendimiento, la capacidad de adaptación, las restricciones, etc.

La arquitectura es un diseño estratégico eso quiere decir, que no se entra en detalle, en la arquitectura menos es más, siempre y cuando muestren todo lo que tienen que mostrar

Modelo: descripción completa de un sistema desde una perspectiva particular y a un nivel de abstracción específico.

Punto de vista: Es una definición teórica de la vista, que es lo que yo quiero mostrar en esa vista que estoy construyendo, para que el que lo vaya a leer interprete cuál es la intención de lo que se está modelando es esta vista en particular. Donde cada vista tiene su punto de vista

Vista: Una vista es una proyección del sistema desde una perspectiva en particular, se puede construir tomando elementos desde un modelo (modelo de análisis, Req, diseño, etc) o de varios. Pensado para un interesado en particular, se construye para satisfacer necesidades de información.
Cuando yo elijo qué elementos quiero mostrar para un interés en particular (perspectiva), al ser imposible mostrar todos los detalles de un sistema en una vista, siendo más lo que oculta que lo que muestra.
Debe ser lo más simple posible, ya que sólo se centran en componentes representativos de la arquitectura.



No hay un conjunto de vistas arquitectónicas estándar sino, trabajamos con el enfoque de philip kruchen que se aplica al proceso unificado de desarrollo

Este enfoque usa diagramas de uml para construir las vistas y para cada una de las 5 vistas tiene un enfoque estático y un enfoque dinámico.

Tipos de vistas: conjunto de vistas que se concilian unas con otras.

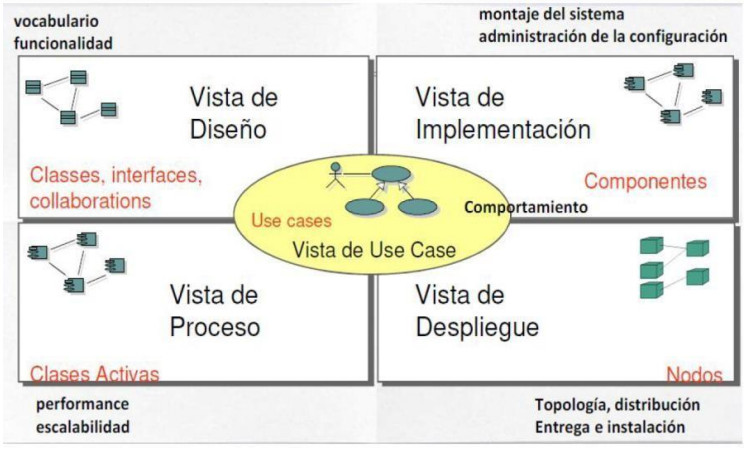
- *Vista de módulo:* contiene vistas de los elementos que se pueden ver en tiempo de compilación. Vista estática de cómo organizamos el código. Definiciones de tipos de componentes, puertos y conectores, clases e interfaces.(Patrón Layered)
  • diagrama de módulo
  • diagramas de casos de uso
- *Vista de ejecución:* contiene vistas de los elementos que se pueden ver en tiempo de ejecución. Incluye escenarios de funcionalidad, listas de responsabilidad y ensambles de componentes, instancias de componentes, conectores y puertos. (Patrones: Publish-Suscribe, Patrón Broker, Messaging, ProcessCoordinator)
  • diagrama de contexto del sistema
- *Vista de distribución:* contiene vistas de elementos relacionados con la distribución del SW en el HW. (Patrones: N- Tier, Mirrored, Farmy Rack) Incluye elementos ambientales.
  • diagrama de distribución
- *Vista de spanning:* contiene cuestiones de confiabilidad, seguridad, performance, escenarios de atributos de calidad, concesiones, ventajas y desventajas.

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

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

No hay producto si no hay arquitectura, y las vistas que tenemos que modelar va a depender del producto que necesita

Modelo"4+1" en el PUD: busca describir y comprender la arquitectura de un sistema a partir de cuatro vistas que se encuentran anidadas por un conjunto de CU SPA en la ***vista arquitectónica de la funcionalidad o vista de casos de uso.*** La descripción del sistema se separa en vistas: cada vista representa una proyección de la descripción completa del sistema, enfatizando un aspecto particular. Cada una puede existir por sí misma, así diferentes usuarios pueden centrarse en las cuestiones de la arquitectura del sistema que más les interesen. Estas 5 vistas interactúan entre sí.



- *Vista Arquitectónica de la Funcionalidad (CU):* Comprende los CU que describen el **comportamiento** del sistema tal como es percibido por los actores (usuarios finales, analistas y encargados de las pruebas).
  Es la primera que aparece ya que elegimos los casos de uso significativos para la arquitectura y de esa vista tenemos que construir y deducir las demás por lo que reforzamos la idea de que es conducido por caso de uso y que se va condicionando la funcionalidad con la arquitectura, por lo que está centrado en la arquitectura.
  Esta vista tiene entre el 10 y el 15 por ciento de los casos de usos totales del sistema, ya que en general alcanza para mostrar las decisiones arquitectónicas que queremos tomar con este producto.

  Diagramas UML:
  - Estática: diagrama de CU, se usa para modelar la parte estática de la funcionalidad.
  - dinámica: *diagramas de interacción (secuencia y comunicación*). El de comunicación se queda escaso en recursos de modelado.

- *Vista de Diseño (lógica):* Identificamos subsistemas y dentro de cada uno de ellos identificamos los componentes que son significativos para la arquitectura y ofrecen sus servicios a través de interfaces, (provista y requerida).

  Propósito: Mostrarnos cuales son los elementos relevantes de la arquitectura que este sistema va a manejar, es decir los elementos que nos van a ayudar a resolver la funcionalidad que está marcada en la vista de la funcionalidad.

  Vemos que componentes importantes de software vamos a necesitar y organizar en subsistemas, ayudándonos con el enfoque del patrón layered de manera que lo que es significativo para la vista funcional este cubierto.

  Los subsistemas se comunican a través de sus interfaces.

  Diagramas UML:
  - estática: diagrama de clases y diagrama de componentes
  - dinámica: diagramas de interacción (*secuencia*)

- *Vista de Procesos (interacción):* muestra el flujo de control entre las partes del sistema (**clases activas**), incluyendo mecanismos de concurrencia y sincronización. No es necesaria hacerla siempre
  Propósito: Mostrar cómo pensamos resolver estos requerimientos, de performance, concurrencia, escalabilidad.
  Usamos esta vista cuando el sistema tiene algunos requerimientos importantes, críticos sobre performance, concurrencia, escalabilidad.

  Diagramas UML:
  - estática: *diagrama de componentes*
  - dinámica: diagrama de secuencia

- *Vista de Implementación (componentes):* comprende los artefactos (**componentes** y módulos de implementación) que se utilizan para ensamblar y poner en ejecución el sistema físico, junto a sus dependencias (código, fuente, librerías).

  Propósito: Toma las decisiones de cómo vamos a estructurar el **repositorio** donde van a contener los componentes de código que se van a ir construyendo y cómo se va a manejar el control de cambio y la configuración de sus componente de código, incluyendo las configuraciones de los ambientes de desarrollo y las políticas de gestión de configuración.

  Diagramas UML:

- estática: diagrama de componentes
- dinámica: *secuencia*

➢ *Vista de Despliegue:* describe la distribución física del sistema, los niveles de HW y los componentes de SW en los **nodos** de HW (computadoras, dispositivos) que forman la **topología** de HW sobre la que se ejecuta el sistema.
Los nodos representan elementos de hardware.
Propósito: Hace una distribución física de HW, muestra que hardware vamos a utilizar, cuanto niveles vamos a tener y cómo se van a comunicar estos distintos niveles y que componentes de cada capa de software vamos a alojar en cada nivel de hardware.
Diagramas UML:
- estática: *diagramas de despliegue* (de niveles de HW y de SW en HW)
- dinámica: diagramas de interacción (*secuencia*)

¿Cuántas vistas? no todos los sistemas requieren todas las vistas:
- **procesador único**: saltear la vista de **despliegue**, esta es imprescindible cuando se crea un producto de software que va a funcionar en un único elemento de hardware.(misma pc, teléfono).
- **procesos simples**: saltear vista de **proceso**, cuando no hay , requerimientos de performance o de escalabilidad, redundancia,esta vista puede ser imprescindible.
- **programas muy pequeños**: saltear la vista de **implementación**

otras vistas que se pueden agregar: vista de los **datos**, vista de **seguridad**, esta vista es utilizada cuando se necesita persistencia de datos, en bases que no son orientados a objetos es decir relacionales.

DOCUMENTACIÓN DE LA ARQUITECTURA

El documento arquitectónico incluye las decisiones tomadas en el proceso de diseño de la arquitectura, siempre sujetas al contexto en el que se tomaron, representadas mediante Vistas Arquitectónicas. Importancia:

- Expresión del sistema y su evolución.
- Permite comprender el diseño al retomarlo luego de un tiempo.
- Permite la comunicación entre todos los involucrados.
- Evaluación y comparación de la arquitectura de una forma consistente.
- Planificación, administración y ejecución de las actividades del desarrollo del sistema.
- Expresión de las características persistentes y soporte de los principios que guían un cambio aceptable.
- Verificación de la implementación del sistema.
- Destaca las decisiones iniciales del diseño que impactarán todo el trabajo de ingeniería de SW que sigue.
- Permite hacer un análisis del diseño.
- Constituye un modelo pequeño y comprensible de cómo está estructurado el sistema y como trabajan sus componentes.

Unidad Nro. 4: Implementación, Prueba y Despliegue en el Proceso Unificado (tp)

WORKFLOW DE IMPLEMENTACIÓN

Se comienza con el resultado del diseño e implementamos el sistema en términos de componentes (archivos de código fuente, scripts, archivos de código binario, ejecutables y similares).

Propósitos:

- desarrollar la arquitectura el sistema como un todo (principal)
- planificar las integraciones de sistema necesarias en cada iteración, siguiendo un enfoque incremental, dando lugar a un sistema que se implementa en una sucesión de pasos pequeños y manejables.
- distribuir el sistema asignando componentes ejecutables a nodos en el diagrama de despliegue.
- implementar las clases y subsistemas encontrados durante el diseño.
- probar los componentes individualmente e integrarlos, compilándolos y enlazándolos a ejecutables.

Papel en el ciclo de vida del SW: es el centro en las iteraciones de construcción; también se hace trabajo de implementación en la fase de elaboración, para crear la línea base ejecutable de la arquitectura, y en la fase de transición, para tratar defectos tardíos.

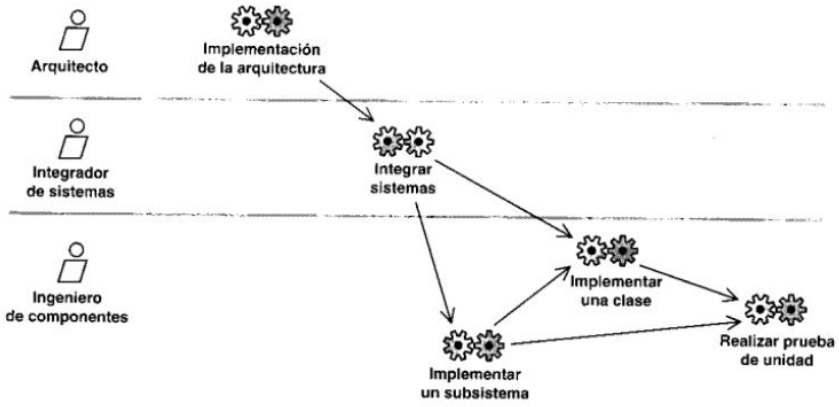
Artefactos

- *Modelo de implementación:* es el resultado principal. Describe cómo se implementan los elementos del modelo de diseño en términos de componentes, y cómo se organizan estos según los mecanismos de estructuración y modularización disponibles en el entorno de implementación y en el lenguaje utilizado, y sus dependencias.
- *Componente:* es el empaquetamiento físico de los elementos de un modelo.
  - <<executable>> programa que puede ser ejecutado en un nodo
  - <<file>> fichero que contiene código fuente o datos
  - <<library>> librería estática o dinámica
  - <<table>> tabla de BD
  - <<document>> documento
- *Subsistema de implementación:* proporcionan una forma de organizar los artefactos del modelo de implementación en trozos más manejables. Puede estar formado por componentes, interfaces, y otros subsistemas (recursivamente). Se manifiesta a través de un "mecanismo de empaquetamiento" concreto en un entorno de implementación determinado (paquete en Java, proyecto en Visual Basic). Los subsistemas de

implementación se corresponden con los subsistemas de diseño en el modelo de diseño, debiendo seguir su traza. El subsistema debería definir dependencias análogas hacia otros subsistemas de o interfaces.

- *Interfaz:* especifican las operaciones implementadas por componentes y subsistemas de implementación. Un componente que implementa (y proporciona) una interfaz debe implementar correctamente todas las operaciones definidas por la interfaz.
- *Descripción de la arquitectura:* contiene la vista del modelo que representa sus artefactos SPA:
  - La descomposición del modelo en *subsistemas*, sus *interfaces* y las *dependencias* entre ellos.
  - *Componentes clave*: los que siguen la traza de las clases de diseño SPA, los ejecutables y los que son generales, centrales, que implementan mecanismos de diseño genéricos de los que dependen otros componentes.
- *Plan de integración de construcciones:* es importante construir el SW incrementalmente en pasos manejables (iteraciones), donde cada uno tiene como resultado una versión ejecutable del sistema llamada *construcción*. Cada una es sometida a pruebas de integración antes de que se cree otra. Para prepararse ante un fallo se lleva un control de versiones para poder volver atrás a una construcción anterior. *Un plan de integración de construcciones describe la secuencia de construcciones necesarias en una iteración:*
  - la funcionalidad que se espera que sea implementada en dicha construcción.
  - las partes del modelo de implementación que están afectadas por la construcción.

Actividades y trabajadores



- ***Implementación de la arquitectura:*** esbozar el modelo de implementación y su arquitectura mediante:
  - La identificación de componentes SPA.
  - La asignación de componentes a los nodos en las configuraciones de redes relevantes.

Arquitecto: es responsable de la integridad del modelo y asegura que es un todo correcto (implementa la funcionalidad descrita en el modelo y en los requerimientos, y solo esa), completo y legible. Es responsable de la arquitectura del modelo, de la existencia de sus partes SPA. También es responsable de asignar componentes ejecutables a nodos, lo que se representa en la vista de la arquitectura del modelo de despliegue. Inician el proceso esbozando los componentes clave en el modelo de implementación. *El arquitecto mantiene, refina y actualiza la descripción de la arquitectura y las vistas de la arquitectura de los modelos de implementación y de despliegue.*

- ***Integrar el sistema:***
  - Integrar cada construcción antes de las pruebas de integración.

Integrador de sistemas: planifica la secuencia de construcciones necesarias en cada iteración y la integración de cada construcción cuando sus partes fueron implementadas, dando lugar a un plan. Para cada construcción, describe la funcionalidad que debería implementarse y qué partes del modelo se verán afectadas. Integra los nuevos componentes en una construcción y la pasa a los ingenieros de pruebas de integración.

- *Planificación de una construcción:* crear un plan de integración de construcciones que describa las necesarias en una iteración y sus requerimientos. Se discute como planificar los contenidos de una construcción.
  Criterios para crear una construcción:
- no debería incluir demasiados componentes nuevos o refinados, sino es muy difícil integrar la construcción y llevar a cabo las pruebas de integración.
- debería estar basada en la construcción previa, y añadir funcionalidad implementando CU completos o escenarios, expandiéndose hacia arriba y hacia los lados en la jerarquía de subsistemas.

Los resultados deberían estar recogidos en el plan y ser comunicados a los ingenieros de componentes responsables de los subsistemas y componentes afectados.

- *Integración de una construcción* recopilando las versiones correctas de los subsistemas de implementación y de los componentes, compilándolos y enlazándolos para generar una construcción.

❖ ***Implementar un subsistema:*** asegurar que cumple su papel en cada construcción, tal y como se especifica en el plan. Se asegura que los requerimientos en la construcción y que afectan al subsistema son implementados correctamente dentro del subsistema.

Ingeniero de componentes: define y mantiene el código fuente de los componentes, garantizando que implementan la funcionalidad correcta. Mantiene la integridad de los subsistemas, garantizando que sigua la traza a los subsistemas de diseño: que sus contenidos y dependencias sean correctos, y que implementan correctamente las interfaces que proporcionan. Implementan los requerimientos sobre los subsistemas y componentes en la construcción. Llevan a cabo las pruebas individuales de cada componente y los pasan al integrador. Se asegura de probar todo el código durante las pruebas de estructura: que cada sentencia ha de ser ejecutada al menos una vez.

❖ ***Implementar una clase de diseño (ing. de componentes):*** en un componente fichero.
- *Esbozo de los componentes* fichero según el tipo de modularización y convenciones de los lenguajes.
- *Generación de código a partir de una clase de diseño:* sólo se genera la signatura (firma) de las operaciones.
- *Implementación de operaciones (métodos):* se elige el algoritmo y estructuras de datos apropiadas, y la codificación de las acciones requeridas por el algoritmo.
- *Los componentes deben proporcionar interfaces apropiadas:* las mismas que las clases diseño que implementa.

❖ ***Realizar prueba de unidad (ing. de componentes):*** probar los componentes implementados como unidades individuales, con pruebas de caja negra y blanca.

WORKFLOW DE PRUEBA

En este WF de trabajo se verifica el resultado de la implementación probando cada construcción, incluyendo tanto construcciones internas como intermedias, así como las versiones finales del sistema a ser entregadas al cliente.

Propósitos:

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

Papel en el ciclo de vida del SW: se centra en las fases de elaboración, cuando se prueba la línea base ejecutable de la arquitectura, y de construcción cuando el grueso del sistema está implementado. Durante la fase de transición el centro se desplaza hacia la corrección de defectos durante los primeros usos y las pruebas de regresión.

Artefactos de la prueba

- *Modelo de pruebas:* describe principalmente cómo se prueban los componentes ejecutables en el modelo de implementación con pruebas de integración y de sistema. Puede describir también como han de ser probados aspectos específicos del sistema (si la interfaz de usuario es utilizable y consistente o si el manual de usuario del sistema cumple su cometido). El modelo de pruebas es una colección de casos de prueba, procedimientos de prueba y componentes de prueba.
- *Caso de prueba:* específica la forma de probar el sistema, incluyendo la entrada o resultado con la que se ha de probar y las condiciones bajo las que ha de probarse.
  Casos de prueba comunes:
  * Un caso de prueba que especifica cómo probar un CU o un escenario específico de un CU (verificación del resultado de la interacción ente actores y el sistema, si se cumplen las precondiciones y postcondiciones y que se siguen la secuencia de acciones especificadas en el CU). Prueba de sistema como "caja negra": prueba del comportamiento observable del sistema.
  * Un caso de prueba que especifica cómo probar una realización de CU de diseño o un escenario específico de la realización (verificación de la interacción entre componentes que implementan el CU). Prueba de sistema como "caja blanca": prueba de la interacción interna entre los componentes del sistema.

  Otros casos de prueba para probar el sistema como un todo:
  - Pruebas de instalación: verifican que el sistema puede ser instalado en la plataforma del cliente y que funcionará correctamente cuando sea instalado.
  - Pruebas de configuración: verifican que el sistema funciona correctamente en diferentes configuraciones.
  - Pruebas negativas: intentan provocar que el sistema falle para poder así revelar sus debilidades. Consiste en utilizar el sistema en formas para los que no sido diseñado ("usar mal el sistema a propósito").
  - Pruebas de tensión o de estrés: identifican problemas con el sistema cuando hay recursos insuficientes o cuando hay competencia por recursos.
- *Procedimiento de prueba:* Un procedimiento de prueba específica como realizar uno a varios casos de prueba o partes de estos. El cómo llevar a cabo un caso de prueba puede ser especificado por un procedimiento de prueba, pero es a menudo útil reutilizar un procedimiento de prueba para varios casos de prueba y reutilizar varios procedimientos de prueba para un caso de prueba.
- *Componente de prueba:* Un componente de prueba automatiza uno o varios procedimientos de prueba o parte de ellos. Los componentes de prueba pueden ser desarrollados utiliza un lenguaje de guiones o un lenguaje de programación, o puede ser grabados con una herramienta de automatización de pruebas.Se utilizan para probar los componentes en el modelo de implementación, proporcionando entradas de prueba, controlando y monitorizando la ejecución de los componentes a probar y, posiblemente, informando de los resultados de las pruebas.
- *Plan de prueba:* describe las estrategias, recurso y planificación de la prueba. La estrategia de prueba incluye la definición del tipo de pruebas a realizar para cada iteración y sus objetivos y el nivel de cobertura de la prueba, código necesario y el porcentaje que deberían ejecutarse con un resultado especifico.
- *Defecto:* es una anomalía en el sistema, puede ser utilizado para localizar cualquier cosa que los desarrolladores necesitan registrar como síntoma de un problema en el sistema que ellos necesitan controlar y resolver. Sirven como retroalimentación tanto para otros flujos de trabajo, como el diseño y el de implementación, como para los ingenieros de pruebas para que lleven a cabo una evaluación sistemática de los resultados de las pruebas.
- *Evaluación de prueba:* es una evaluación de los resultados de los esfuerzos de prueba (cobertura del caso de prueba, la cobertura de código y estado de los defectos).

Actividades y trabajadores:

- *Planificar prueba (ing. de pruebas):* planificar los esfuerzos de prueba de cada iteración. Se crea el plan de prueba estableciendo las estrategias y recursos.
  - Describir una estrategia de prueba: casos y procedimientos.
  - Estimar los requerimientos para el esfuerzo de la prueba (ejemplo: recursos humanos y sistemas necesarios).
  - Planificar el esfuerzo de la prueba.
- *Diseñar prueba:* los ingenieros de pruebas de integración y de sistema prueban cada construcción y detectan cualquier defecto que encuentren en ellos.

- Identificar y describir los casos de prueba para cada construcción (de sistema, de integración, de regresión).
- Identificar y estructurar los procedimientos de prueba especificando cómo realizar los casos de prueba: intenta reutilizar procedimientos de prueba existentes.

  ** Diseñador de pruebas:* es responsable de la integridad del modelo de pruebas, asegurando que cumple con su propósito. Deciden los objetivos apropiados y la planificación de las pruebas. Seleccionan y describen los casos de prueba y los procedimientos de prueba correspondientes que se necesitan, y son responsables de la evaluación de pruebas de integración y de sistema cuando éstas se ejecutan. Intentan crear procedimientos de prueba que puedan ser reutilizados en varios casos de prueba. Los diseñadores de pruebas evalúan los resultados de la prueba comparando los resultados obtenidos con los objetivos esbozados en el plan de prueba. Éstos preparan métricas que les permiten determinar el nivel de calidad del software y qué cantidad de pruebas es necesario hacer.

- *Implementar prueba:* automatizar los procedimientos de prueba creando componentes de prueba, si es posible.
  ** Ingeniero de componentes:* es responsable de los componentes de prueba que automatizan algunos procedimientos. Crean a continuación los componentes de prueba para automatizar algunos procedimientos de prueba.

- *Realizar pruebas de integración:* necesarias para cada una de las construcciones creadas en una iteración y se recopilan los resultados.
  ** Ingeniero de pruebas de integración:* son los responsables de realizar las pruebas de integración que se necesitan para cada construcción producida en el flujo de trabajo de la implementación y de documentar los defectos en los resultados.

- *Realizar prueba de sistema:* puede empezar cuando las pruebas de integración indican que el sistema satisface los objetivos de calidad de integración fijados en el plan de pruebas de la iteración actual.
  ** Ingeniero de pruebas de sistema:* es responsable de realizar las pruebas de sistema necesarias sobre una construcción que muestra el resultado (ejecutable) de una iteración completa y de documentar los defectos en los resultados.

- *Evaluar la prueba (ing. de pruebas):* evaluar los esfuerzos de prueba de una iteración.

Prueba de Software: proceso en el cual se prueba el producto para corroborar que cumpla con la ERS y también detectar errores imprevistos y comprobar que el sistema satisface los RNF.

Niveles en los que se puede probar el software:

- *Prueba de unicidad:* se involucran unidades de programa, clases de objetos, bloques, paquetes de servicios. Se enfocan en comprobar la funcionalidad de objetos y métodos. Es el nivel de prueba más bajo y normalmente lo hace el mismo desarrollador. Se realiza en el WF de implementación.
  - *Prueba de caja negra (de especificación):* verifican el comportamiento de la unidad observable

externamente, la interfaz de la unidad, lo que hace sin importar cómo lo hace. Además de verificar si se producen salidas, hay que verificar si son correctas.

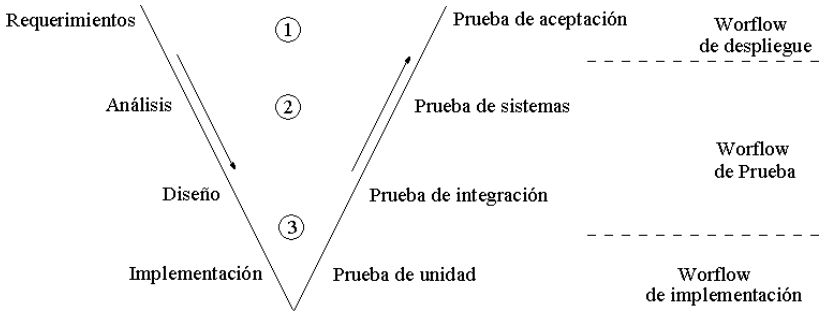
- *Prueba de caja blanca (de estructura):* se verifica si la estructura interna es la correcta. Todos los caminos posibles planteados en el *código* deben ser contemplados y ejecutados. Por lo general se realiza al último.
- *Prueba basada en estados:* prueba la interacción entre operaciones de una clase, monitoreando los cambios en los atributos de los objetos. Se basa en diagramas de transición estados, así cada estado y cada transición son atravesados al menos una vez.

  Cuando se pone a prueba las clases de los objetos, hay que brindar cobertura a todas las características del objeto: operaciones, atributos y estado. La generalización o herencia dificulta esto, ya que se debe poner a prueba la operación heredada en todos los contextos en que se utilice. Siempre que sea posible, se deben automatizar las pruebas de unidad (por ejemplo, JUnit).

- *Prueba de Integración (de componentes):* una vez que las unidades fueron certificadas en las pruebas de unidad, deberían integrarse en unidades más grandes y finalmente al sistema. Se verifica que los componentes integrados en una construcción funcionan correctamente juntos. Se incluyen pruebas de paquetes de servicio de CU, subsistemas y el sistema completo. Es necesaria porque al combinar las unidades pueden aparecer nuevas fallas. La mayoría de los casos de prueba de integración pueden ser derivados de las realizaciones de CU de diseño, ya que describen cómo interaccionan las clases y los objetos, y por tanto cómo interaccionan los componentes. El acceso a la funcionalidad de dichos objetos es a través de la interfaz de componente definida, por lo tantos esta prueba se enfoca en mostrar que dicha interfaz se comporta según su especificación.

- *Prueba de Sistema:* una vez probado todos los CU por separado se prueba que el sistema funcione correctamente como un todo completo. Requiere el uso de casos de prueba típicos. Se realiza en el WF de prueba y por personal de testing. Se enfoca en verificar las interacciones entre los actores y el sistema. Prueban principalmente combinaciones de CU instanciados bajo condiciones diferentes (diferentes configuraciones de HW, cargas del sistema, números de actores y tamaños de la base de datos). Las pruebas de sistema demuestran que los componentes son compatibles, que interactúan correctamente y que transfieren los datos correctos en el momento adecuado a través de sus interfaces. Algunos elementos de funcionalidad del sistema sólo se hacen evidentes cuando se reúnen los componentes.

- *Prueba de aceptación:* es la misma prueba que la de sistema, pero el cliente es quien la realiza. Se ejecuta en el WF de despliegue.



Tipos de pruebas se pueden realizar para probar un software a nivel de prueba de sistema o de prueba de aceptación de usuario.

- *Prueba de operación normal:* se prueban los RF del sistema, siguiendo el curso normal.
- *Prueba negativa:* se prueban los RF del sistema con intenciones de hacerlo fallar.
- *Prueba de escala completa:* se prueba el sistema en su máxima capacidad sin sobrepasar sus límites.

- *Prueba de documentación:* es una prueba de consistencia entre el sistema y toda la documentación asociada a él (manual de usuario, de instalación, de configuración).
- *Prueba de proceso de negocio:* se hace sobre los RF, se integran varios CU para armar un circuito de negocio y luego poder testearlo.

WORKFLOW DE DESPLIEGUE

Propósito: volcar el producto finalizado a sus usuarios, presentar el software. Se centra en la fase de transición, donde el usuario final puede probar el sistema para saber cómo trabaja en un ambiente real de trabajo.

Actividades y trabajadores

- *Plan de despliegue:* debe tener en cuenta cómo y cuándo entregar el nuevo software, y también asegurarse que el usuario final tiene toda la información necesaria para recibirlo adecuadamente y comenzar a usarlo. Los planes de despliegue incluyen una prueba de la beta del programa, para ir asesorando al usuario de forma temprana. El planeamiento de despliegue general del sistema requiere un alto grado de colaboración y preparación del cliente.
  - *Gerente de despliegue:* planifica y organiza el despliegue, establece el listado de las actividades y pasos a seguir para desarrollar el WF. Es responsable de la retroalimentación de las beta test asegurando que el producto es empaquetado apropiadamente para su envío.
- *Desarrollar material de soporte:* cubre toda la información que será requerida por el usuario final para instalar, operar, usar y mantener el sistema. También incluye material de capacitación para hacer efectivos todos los usos posibles del sistema.
  Desarrollador de curso: Planea y produce material para capacitación necesario en la formación del usuario final una vez que se entrega el Software.
  Escritor técnico: redacta las especificaciones técnicas del producto y toda la ayuda y el material de soporte para el usuario final del SW.
- *Probar el producto en el lugar de desarrollo:* determina si el producto tiene la madurez suficiente para ser entregado como producto final o como distribución para beta-testers. El testeo se hace para pulir detalles, permitiendo al usuario final retroalimentar y mejorar el producto antes de su entrega final.
  Tester: ejecuta pruebas de aceptación para encontrar fallas y garantizar que fueron suficientes y adecuadas.
- *Crear el producto final:* hay que asegurarse que el producto está preparado para la entrega al cliente. La reléase consiste en todo lo que el usuario final necesitará para instalar y ejecutar el SW finalmente.
  Implementador: encargado de todos los artefactos que ayudaran al usuario final a instalar el producto (scripts de instalación, instaladores, guías).
- *Prueba Beta del producto final:* esto requiere que el producto sea instalado por el cliente, quien proveerá información sobre su performance y usabilidad. Es esencial para asegurarse que las expectativas del cliente fueron satisfechas y la información provista por el usuario, se retroalimente a la próxima iteración del desarrollo.
- *Probar el producto en el lugar de instalación:* el producto debe ser instalado y probado por el cliente. Basándonos en las iteraciones y pruebas anteriores, en esta prueba no deberíamos encontrarnos con sorpresas y debería ser solo una formalidad para que el cliente acepte el sistema.
  Gerente de proyecto: es el intermediario con los clientes. Es responsable de aprobar el despliegue final y las versiones betas, según la retroalimentación y las evaluaciones de resultados; y de la aceptación del cliente sobre el envío del producto.
- *Empaquetar el producto (opc.):* estas actividades opcionales describen lo que debe pasar para producir un producto de "software empaquetado". En este caso, el producto final se guarda como producto maestro para su

producción en masa y luego empaquetado en cajas con la lista de contenidos para su envío al cliente.
Artista gráfico: encargado de todo el material artístico del proyecto que será visible en el packaging o en el sistema mismo.

- *Proveer acceso a sitio de descarga (opc.)*

Artefactos del despliegue: el artefacto clave es el producto terminado (reléase), que puede consistir en:
- El SW ejecutable, en todos los casos.
- Artefactos de instalación: scripts, herramientas, archivos, guías, información de licencia.
- Release notes: notas que describen el producto terminado al usuario final.
- Material de Soporte: como los manuales de usuario, operaciones y mantenimiento.
- Manuales de capacitación

En el caso de productos manufacturados, se requieren artefactos adicionales como:
- La lista completa de materiales a ser incluidos en el producto manufacturado
- Empaquetamiento: contenedor del producto y arte del producto.
- Dispositivos de almacenamiento: el material en el cual el producto será vendido (CDs, DVD, etc)

Otros artefactos usados, pero no necesariamente entregados al cliente son:
- Los resultados de las pruebas.
- Resultados de retroalimentación.
- Resumen de las pruebas de evaluación.

Unidad Nro. 5: Evolución del software

PROCESOS DE EVOLUCIÓN DEL SOFTWARE

La ingeniería de software se debe considerar como un proceso en espiral, con requerimientos, diseño, implementación y pruebas continuas, a lo largo de la vida del sistema. Después de distribuir un sistema, inevitablemente debe modificarse, con la finalidad de mantenerlo útil y que no se vuelva obsoleto.

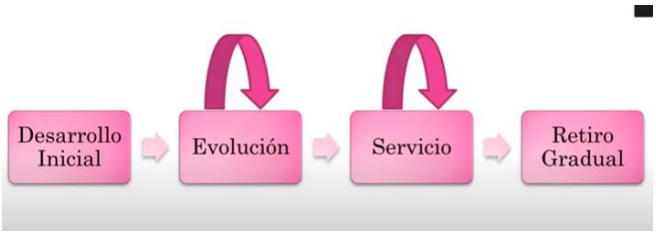
Una vez construido el software y liberado la primera versión de nuestro sistema, hay que empezar a mantener y evolucionar el mismo.

Los procesos de evolución de SW varían dependiendo el tipo de SW que se mantiene, los procesos de desarrollo usados en la organización y las habilidades de las personas que intervienen. Según el tipo de organización, este puede ser un:
- *proceso informal*: las solicitudes de cambio provienen de conversaciones entre usuarios y desarrolladores.
- *proceso formalizado:* con documentación estructurada generada en cada etapa del proceso.

Los procesos de identificación de cambios y evolución del sistema son cíclicos y continúan a lo largo de la vida de un sistema. Las propuestas de cambio son el motor para la evolución del sistema, las cuales provienen de:

* Requerimientos existentes que no se implementaron en el sistema liberado
* Peticiones de nuevos requerimientos: generados por cambios empresariales como las expectativas del usuario
* Reportes de bugs de los participantes
* Nuevas ideas para la mejora del SW por parte del equipo de desarrollo

Ciclo de vida en la evolución del software



- Desarrollo inicial: Definimos un proyecto y cuando se termina, se lo pone en producción y pasa a la siguiente etapa.
- Evolución: Se ingresan nuevos requerimientos, se corrigen errores.
- Servicio: Cuando decidimos no incorporar más requerimientos, solo se hace corrección de errores, le damos mantenimiento.
- Retiro gradual: Cuando no le damos más mantenimiento al sw.

Identificación de cambio y proceso de evolución :

- Es un ciclo: Proceso de identificación de cambio -- Propuestas de cambio -- Proceso de evolución de SW -- nuevo sistema

Proceso de identificación de cambio: Habla de cómo atender las peticiones de los clientes, no importa quién sea el cliente, no hay que darle prioridad a nadie, hay que armar una solicitud de cambio, armando un proceso de evolución, sobre cómo vamos a realizar estos cambios.

Actividades del proceso de evolución - Petición de proceso de cambio:

- *Análisis de cambio:* el *costo* y el *impacto* de los cambios se valoran para saber cuánto resultará afectado el sistema y cuánto costaría implementarlo.
- *Planeación de la versión:* si la propuesta se acepta, se planea una nueva versión del sistema en la que se consideran todos los cambios propuestos (reparación de fallas, adaptación y nueva funcionalidad).
- *Implementación del sistema:* si los desarrolladores del sistema original no son los responsables del cambio, involucra la comprensión del sistema. Se deciden cuáles cambios implementar en la siguiente versión.
- *Liberación a los clientes:* después de implementarse, se valida y libera una nueva versión del sistema. Luego, el proceso se repite con un conjunto nuevo de cambios propuestos para la siguiente liberación.

Leyes de Lehman: aplican para sistemas empresariales relativamente grandes, que requieren que se agregue funcionalidad constantemente para poder seguir brindando valor al negocio.

- *Cambio continuo:* un SW utilizado en un entorno real debe cambiar, sino se vuelve progresivamente menos útil.
- *Complejidad creciente:* a medida que transcurre la vida de un SW y se lo mantiene, su estructura se vuelve más compleja y difícil de cambiar. Deben dedicarse recursos para que mejore continuamente el SW.
- *Evolución del programa:* es un proceso autorregulador. Factores como tamaño y tiempo entre release, así como defectos reportados permanecen constantes.
- *Estabilidad organizacional:* a medida que transcurre la vida del SI, su tasa de desarrollo permanece constante e independiente de los recursos asignados al mismo.
- *Conservación de familiaridad:* en la vida del SI la tasa de cambio con cada release permanece constante.

Estrategias de evolución de SW

	Predicción de **Mantenimiento**	**Reingeniería** de SW	Mantenimiento preventivo mediante **refactorización**
Aplica a	SW recientemente introducido al mercado	SW antiguo, obsoleto, degradado por el paso del tiempo	Cualquier SW, incluso mientras se lo desarrolla
Se utiliza cuando	Mientras se use el SW	El SW se vuelve muy difícil de mantener	Toda la vida del SW
Propósito	Evitar que el SW deje de ser útil y agregar nuevas funcionalidades conforme avanza el tiempo.	Actualizar y mejorar la calidad de un SW muy antiguo y difícil de mantener, para volverlo más mantenible y extenderle la vida útil.	Rever lo ya hecho y mejorarlo constantemente. El objetivo principal es evitar preventivamente el envejecimiento del SW.
Costo	Muy alto, representa normalmente cerca de 70% del costo del SW	Bajo, muchas etapas son automáticas	Si se ejecuta bien, el costo es medio porque si bien es el más caro a corto plazo, a largo plazo se amortiza porque futuros cambios son más fáciles de aplicar.
Riesgo	Alto, el mismo proceso de mantenimiento introduce errores que requieren más mantenimiento y así recursivamente.	Bajo, el foco es no cambiar funcionalidad, y utilizar herramientas automáticas.	Medio, si bien se utiliza acompañado de muchas herramientas para prevenir errores, es imposible que ante tanto cambio de código no se introduzca algún defecto.

Autor: Andermatten – Martinez – Sánchez Díaz

Factores que afectan los costos de la evolución del SW: hacer que el SW sea más fácil de entender y de cambiar puede *reducir los costos de evolución*. Las buenas técnicas de ingeniería de SW, como la especificación precisa (completa documentación y clara definición de variables), el uso de desarrollo OO y la administración de la configuración, contribuyen a *reducir los costos de mantenimiento*. Un factor influyente son los estándares de programación, que ayudan a reducir costos evitando la degradación del SW.

Suele ser más costoso agregar funcionalidad después de que un sistema está en operación, que implementar la misma durante el desarrollo porque la mayoría de las organizaciones *consideran el desarrollo y el mantenimiento como actividades separadas*, provocando que los equipos no sean los mismos, obligando a los de mantenimiento a emplear tiempo en comprender el sistema desarrollado antes de realizar implementaciones. Así no se realizan esfuerzos de desarrollo para crear SW mantenible ya que no fueron contratados para el posterior mantenimiento, haciendo que se emplee personal de mantenimiento inexperimentado y no familiarizado con el dominio.

Por otro lado, conforme se realizan cambios al programa, *su estructura se degrada*, volviéndose más difícil de comprender y cambiar. Las técnicas de reingeniería de SW son aplicables para mejorar la estructura y comprensibilidad del SI.

MANTENIMIENTO DE SOFTWARE

Es el proceso general de cambiar un sistema después de que este se entregó. Los cambios pueden ser corregir errores de codificación, de diseño, o de especificación o incorporar nuevos requerimientos. Puede modificar o agregar funcionalidad al sistema. No se modifica la arquitectura. Puede requerirse por 3 motivos:

- Corregir fallas en el software
- Adaptar el software por una corrección en los requerimientos
- Modificar el software a un nuevo entorno operativo.

Tipos de mantenimiento

- *Correctivo (reparaciones de fallas 17%):* los usuarios detectan fallas cuando el SW ya está en producción, habiendo pasado todas las etapas. Llega un error que no se contempló en los tests. Es el costo de la no calidad.
  Tipos de errores:
  - de codificación: baratos de corregir
  - de diseño son más costosos, ya que pueden implicar la reescritura de muchos componentes
  - de requerimientos son los más costosos, ya que podría necesitarse un extenso rediseño del sistema.

Problemas:

- χ debe realizarse con urgencia
- χ no es cobrable
- χ genera situaciones de malestar con el cliente
- χ hay que compensarlo

❖ *Adaptativo (adaptación ambiental 18%):* todo evoluciona y se demanda eso, no implica nada funcionalmente nuevo, solo cambios a nivel de RNF. Se desea cambiar el ambiente del SW, que funcione en otro dispositivo o entorno, soporte más usuarios, que se pueda usar entre la 1 y las 5 AM, BD más grande, etc.
Problema: tener que pasar todo el proceso de adaptación y actualizar el presupuesto.

❖ *Perfectivo (adición de funcionalidad 65%):* es el que conlleva un mayor esfuerzo de mantenimiento, un mayor presupuesto, y es el más común y más frecuente. Se agregan funcionalidades o cambian las existentes (*cambios en los RF)*. Cambian las políticas y reglas de negocio, disposiciones legales que impactan los procesos de negocio. El cliente demanda agregar cuando descubre que quiere nuevas cosas. Si no generan peticiones de cambio es porque no usan el sistema.

Es difícil diferenciarlos porque muchas veces se mezclan, a veces al adaptar el sistema a un nuevo ambiente se introduce nueva funcionalidad para aprovechar nuevas capacidades del ambiente. O al introducir una nueva funcionalidad se arregla un defecto preexistente. En resumen, los tres tipos de cambio tienen el mismo objetivo, que es *mantener el sistema útil*. Los mantenimientos adaptativo y perfectivo no son atribuibles a algo que hicimos mal, por lo que son *cobrables*. Cuando vienen peticiones de cambio (requerimientos), se debe tener rastreabilidad (trazabilidad). Lo que comparten estos tipos de mantenimientos es:

1) *Análisis de impacto:* qué (requerimiento), cuándo (calendario, fecha límite), cuánto (costo, es lo último que se calcula), esfuerzo

2) *Comunicárselo al cliente.*

Predicción de mantenimiento: se debe tratar de predecir qué cambios deben proponerse al sistema y qué partes del pueden ser más difíciles de mantener. También tratar de estimar los costos de mantenimiento globales para un sistema durante cierto lapso de tiempo. Para predecir el número de peticiones de cambio se requiere un entendimiento de la relación entre el sistema y su ambiente externo, valorando:

1. *El número y la complejidad de las interfaces del sistema:* cuantas más interfaces y más complejas sean, más probable será que se requieran cambios de interfaz conforme se propongan nuevos requerimientos.
2. *El número de requerimientos de sistema inherentemente inestables:* los requerimientos que reflejan políticas y procedimientos de la organización son más inestables que los que se basan en un dominio estable.
3. *Los procesos de negocio donde se usa el sistema:* a medida que evolucionan estos procesos, generan peticiones de cambio del sistema.

REINGENIERÍA DE SOFTWARE

Significa reestructurar o reescribir un sistema heredado sin cambiar su funcionalidad, volver a hacer ingeniería: se crea un nuevo sistema a partir de uno ya existente. Para hacer que los sistemas heredados sean más sencillos de mantener, someten a reingeniería para mejorar su estructura y entendimiento. La reingeniería puede implicar volver a documentar el sistema, refactorizar su arquitectura, traducir los programas a un lenguaje de programación moderno, y modificar y actualizar la estructura y los valores de los datos del sistema. Este proceso no modifica la funcionalidad, y evita hacer grandes cambios en la arquitectura. La reingeniería de SW es un enfoque que tiene el propósito de mejorar la calidad del SW, cómo está hecho, con la intención de que el costo de mantenimiento sea menor. Es útil cuando el sistema es caro de mantener y también es riesgoso alterar su funcionamiento.

Actividades de la reingeniería: no necesariamente se requieren todas

1. *Traducción del código fuente:* el programa se convierte de un lenguaje de programación antiguo a una versión más moderna, o a un lenguaje diferente.

2. *Ingeniería inversa:* el programa se analiza y se extrae información del él. Suele ser un proceso automatizado.
3. *Reestructurar el SW:* la estructura de control se analiza y modifica para facilitar su lectura y comprensión. Puede ser automatizado con alguna intervención manual.
4. *Modularizar el programa:* las partes relacionadas se agrupan y, donde es adecuado, se elimina la redundancia. En algunos casos, implica refactorización arquitectónica. Es un proceso manual.
5. *Reingeniería de datos:* los datos procesados por el programa cambian para reflejar cambios al programa. Significa la redefinición de los esquemas de BD y convertir las BD existentes a la nueva estructura.

Ventajas de la reingeniería frente a la sustitución

- *Reducción del riesgo:* pueden cometerse errores en la especificación del sistema o problemas de desarrollo, y las demoras o problemas de la introducción del nuevo software sustituto podrían significar costos elevados.
- *Reducción de costos:* el costo de la reingeniería puede ser significativamente menor que desarrollar SW nuevo.

Desventajas

- χ no se puede convertir un sistema funcional a un sistema OO.
- χ grandes cambios no pueden hacerse de forma automatizada.
- χ los sistemas con reingeniería no serán tan mantenibles como los nuevos desarrollados con técnicas modernas.

REFACTORIZACIÓN

Es el proceso de modificar un programa para frenar la degradación mediante el cambio, mejorar su estructura, reducir su complejidad o hacerlo más fácil de entender. Mientras se refactorice un programa no se debe agregar funcionalidad, sino que hay que concentrarse en la mejora del programa.

Refactorización vs reingeniería

- χ *Reingeniería:* se lleva a cabo después de haber mantenido un sistema durante cierto tiempo. Se crea un nuevo sistema a partir de un sistema heredado utilizando herramientas automatizadas.
- χ *Refactorización:* es un proceso continuo de mejoramiento debido al proceso de desarrollo y evolución. Intenta evitar la degradación de la estructura y el código, ya que aumentan los costos y las dificultades por mantenerlo.

Mejoras de la refactorización

1. *Código duplicado:* se descarta o se implementa como un solo método.
2. *Métodos largos:* deben rediseñarse en varios métodos más pequeños.
3. *Sentencias case (switch):* en un lenguaje OO se pueden reemplazar mediante polimorfismo.
4. *Aglomeración de datos:* ocurre cuando el mismo grupo de objetos de datos vuelven a ocurrir en muchos lugares en un programa. Pueden sustituirse con un objeto que encapsule todos los datos.
5. *Generalidad especulativa:* ocurre cuando los desarrolladores incluyen generalidad en un programa, en caso de que se requiera en el futuro. Puede simplemente eliminarse.