Diseño de Sistemas

Resumen Teórico 2021

Por Santiago D. Pérsico

ÍNDICE

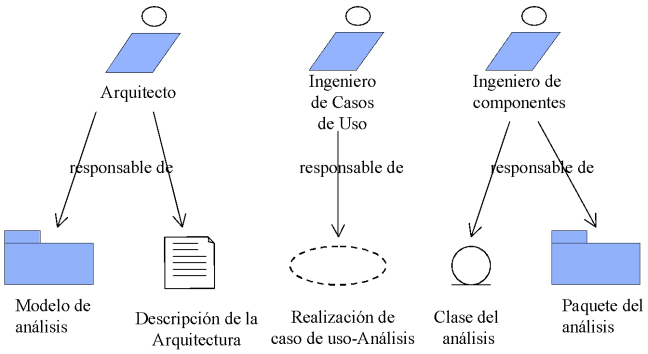
WF de Análisis

Como los errores en el WF de Diseño e Implementación son caros de cometer, utilizamos el WF de Análisis para captar los requerimientos reales y específicos del sistema, para entender de forma precisa, y detallada, pero conceptual, de que es lo que el sistema tiene que hacer.

En el **Ciclo de Vida de Software**, lo encontramos al final de la fase de **Inicio** (ayudando a planificar y definir objetivos, solapandose mucho con la captura de requisitos) y su foco está en la de **Elaboración** (facilitando la comprensión de los requisitos, para lograr una arquitectura robusta y estable).

En él se estudian los requisitos obtenidos en la captura de requisitos, se los refina y estructura, utilizando términos más técnicos/específicos, y con un modelo de objetos conceptual, se razona y analiza la estructura interna del sistema.

El flujo se forma por clases y paquetes que se relacionan, y definen Realizaciones de CU de una manera más cercana a la del desarrollador.



Artefactos

- **(ARQ) Modelo Análisis**: es el que se representa por un Sistema de Análisis, es decir; el paquete de mayor nivel del sistema, del cual se descomponen los paquetes y clases de análisis para descomponer el sistema en subsistemas. Está siempre en lenguaje de Negocio, captura la idea general, y distingue el problema de la solución.
  *En eso se diferencia del Modelo de CU, **el MA es más orientado al desarrollador que al cliente**, por lo que tiene detalles más técnicos, orientados al diseño, desarrollo y mantenimiento del sistema. Vista Interna(Análisis) vs Externa(CU).*
- **(COMP) Clases de Análisis**: son abstracciones C/U de un elemento del sistema, que se centran en los requerimientos funcionales, son abstractas y claras en su definición y en sus responsabilidades.
  - **Sobre las Clases de Análisis:**
    - Su nombre es obligatorio y debe ser descriptivo
    - Deberían tener pocas responsabilidades por clase.
    - No deben ser ni muy grandes ni muy pequeñas
    - No debería haber árboles de herencia muy profundos.

  Y pueden encajar en uno de los 3 estereotipos:
  - **Interfaz**:
    - Modelan las interacciones entre el actor y el sistema.
    - Aíslan cambios de la UI en una o más clases de interfaz.
    - Representan abstracciones de interfaz (ventanas, botones, etc.)
  - **Entidad**:
    - Se usan para modelar datos/información de mayor persistencia/vida (candidatas).
    - Son derivadas del MODP (generalmente).
  - **Control**:
    - Representan Control, Lógica, Secuencia, Coordinación de un procedimiento, y se usan para encapsular el control de un CU concreto. Se pueden desglosar para reducir la complejidad.
    - Modelan los aspectos DINÁMICOS del sistema, porque manejan/coordinan las acciones del mismo.

- **(CU) Realización de CU**: es una descripción del procedimiento en términos de las clases participantes en el mismo, y los objetos y sus interacciones en una traza de sucesos de un CU.

  Se componen por una descripción textual del flujo de sucesos, diagramas de clases de análisis, y diagramas de interacción, estos últimos pueden ser:

  - **Diagrama de Comunicación**: que destaca la organización de los objetos que participan en una interacción, mediante un grafo en el que los objetos son nodos, los enlaces son arcos, y los mensajes son "adornos numerados" sobre los arcos entre los nodos. Para cada CU.
    - **Mensaje**: comunicación entre dos objetos en una interacción.
      - **Iteración**: Se muestra mediante "*"
  - **Diagrama de Secuencia**: es como el DC, enfatizan la secuencia de mensajes ordenada en el tiempo, y muestran más fácilmente la colaboración entre objetos. Muestra las interacciones entre líneas de vida como una secuencia ordenada en el tiempo de eventos. Permiten ver claramente el momento de creación de un objeto, la activación de un objeto, y permiten adjuntar documentación mediante notas.
    - Fragmentos combinados y operadores: todo fragmento combinado tiene un operador, uno o más operandos y cero o más condiciones de protección. *Es como una combinación de operaciones, agrupadas de una manera lógica.*
  - **Diagrama de Visión de Interacción**. En él, los nodos hacen referencia a otras Actividades.
  - **Diagrama de Tiempo**. Enfatizan los aspectos en tiempo real de una interacción.
- **(COMP) Paquete de Análisis**: Es un elemento de Agrupación de UML2, con un espacio de nombres encapsulado, que cumple el papel de contenedor de elementos del modelo. Se usan para organizar artefactos del modelo (como clases de análisis, Realizaciones de CU y otros paquetes de análisis) y encapsular elementos del Modelo. Los paquetes deberían ser Altamente Cohesivos, y con un Bajo Acoplamiento. Se pueden anidar paquetes.

- **(ARQ) Descripción de la Arquitectura:** es una vista de la arquitectura del modelo de análisis, donde se describe la descomposición del modelo en paquetes de análisis y sus dependencias, las clases de análisis fundamentales, y las realizaciones de CU más importantes o críticas.

Actividades

- **(ARQ) Análisis de la Arquitectura:** se busca obtener un esbozo del modelo de análisis, y la arquitectura, identificando los paquetes de análisis (en base a los requisitos funcionales), las clases de entidad más obvias (deduciendo de las clases de dominio), y los requisitos especiales comunes (consideraciones, por ejemplo, de persistencia o seguridad).
- **(CU) Análisis de Caso de Uso:** en esta actividad, se busca identificar las clases cuyos objetos serán necesarios para llevar a cabo el flujo de sucesos, distribuir el comportamiento y capturar requisitos especiales
  - Identificando las clases de Análisis (Entidad, Interfaz y Control)
  - Describiendo las interacciones entre objetos de análisis del CU (con Diagramas de Colaboración).
- **(COMP) Analizar una Clase:** es decir, sus atributos, responsabilidades y relaciones entre ellas.
- **(COMP) Analizar un Paquete:** para garantizar que el paquete tenga Alta Cohesión y Bajo Acoplamiento, que cumpla su objetivo y para describir las dependencias del mismo con otros paquetes.

Patrones GRASP

Hay dos tipos de responsabilidades, **conocer** (datos o métodos propios, o de objetos relacionados) y **e** (acciones que él mismo puede hacer). Una responsabilidad no es lo mismo que un método, las responsabilidades se implementan por medio de métodos que no actúan solos o colaboran con otros métodos u objetos.

Los Patrones GRASP describen los principios fundamentales de diseño de objetos y asignación de responsabilidades al momento de diseñar la colaboración entre objetos y clases.

Un Patrón es una descripción de un problema y la solución, a la que se le da un nombre, y que se puede aplicar a nuevos contextos. Proporciona consejos sobre cómo aplicarlo en varias circunstancias, y considera los pros y contras de hacerlo.

Lo importante de un patrón no es expresar ideas nuevas de diseño, los patrones codifican conocimientos y principios existentes para evitar "reinventar la rueda", aplicando soluciones que ya fueron probadas con éxito.

Experto en Información: nos indica que debemos asignar las responsabilidades a la clase que cuente con la información necesaria para poder cumplirlas. Poner el comportamiento lo más cerca posible de los datos.

Creador: indica que la responsabilidad de crear a un objeto debe recaer sobre el objeto que agregue, utilice, contenga, o registre, o contenga datos de inicialización del objeto a crear.

Manejador: nos indica que debemos asignar la responsabilidad de controlar los eventos del sistema a una clase que represente al sistema global(o dispositivo, o subsistema) o que maneje exclusivamente a un caso de uso.

Bajo Acoplamiento: insta a que las clases sean lo más independiente y modular posible, buscando reutilización, evitando el uso de referencias, y manteniendo el nivel de conocimiento de cada clase al mínimo.

Alta Cohesión: Cohesión es la fuerza con la que se relacionan los objetos y el grado de focalización de las responsabilidades de un objeto. Es decir,, se busca que cada clase tenga una importante funcionalidad relacionada y poco trabajo por hacer.

No Hables Con Extraños: indica que los mensajes se deben enviar entre objetos con relación directa, nunca indirecta. Establece restricciones a los mensajes a mandar y a quien se puede hacerlo.

Polimorfismo: Siempre que se tenga que llevar a cabo una responsabilidad que dependa del tipo, se tiene que hacer uso de polimorfismo al asignar el mismo nombre a servicios en diferentes objetos.

Fabricación Pura: La fabricación pura se da en las clases que no representan un ente u objeto real del dominio del problema, sino que se ha creado intencionalmente para disminuir el acoplamiento y aumentar la cohesión.

Variación Protegida: todo lo que prevemos que es susceptible de modificaciones, lo envolvemos en una interfaz, utilizando el polimorfismo para crear varias implementaciones y posibilitar implementaciones futuras, de manera reducir el acoplamiento y aumentar el encapsulamiento.

UML (Unified Modeling Language)

Un modelo es representación simplificada de la realidad utilizada, en nuestro caso, para comprender un sistema, para **ver** cómo es, definir cómo **construirlo**, si aún no lo hicimos, **especificar** su estructura y cómo funciona, y **documentar** su arquitectura y las decisiones que se tomaron con respecto al sistema.

UML es un lenguaje (*con un vocabulario y reglas semánticas*) estándar de modelado (*que se enfoca en la representación conceptual y física*) para realizar modelos de software, independiente del proceso de desarrollo, aunque dirigido a procesos basados en Casos de Uso, centrados en la arquitectura, y que sean iterativos e incrementales.

UML sirve para Visualizar, Especificar, Documentar y Construir modelos.

Los lenguajes de modelado surgieron en los 70 fragmentados, y se fueron unificando y estandarizando, y en la actualidad utilizamos **UML 2.5**, aunque se continúan desarrollando.

UML es unificado porque sirve durante todo el ciclo de vida de desarrollo, porque puede modelar cualquier tipo de sistemas, porque es una unificación de los lenguajes de modelado anteriores de la historia, y porque no depende del lenguaje de desarrollo, ni su plataforma, ni del proceso de desarrollo. UML tiene sus propios conceptos internos, lo que lo hace **independiente**, **coherente** y **uniforme** en su aplicación.

Bloques de Construcción de UML

- **Elementos**

  - **Elementos Estructurales:** partes estáticas, representan conceptos o cosas materiales.

  Clase: representa un elemento del dominio, sus atributos y responsabilidades. (+ Clase activa)
  Interfaz: Define comportamientos para una clase o componente, las operaciones que esta puede realizar, sin especificar su implementación. *Separa funcionalidad de Implementación.*
  Colaboración: representa una interacción de objetos para lograr una funcionalidad.
  Componente: es un conjunto de funcionalidades (servicio encapsulado)
  Artefacto (UML2): es una parte física del sistema (archivo fuente, tabla de DB, otros).
  Nodo: Representa un elemento de Hardware (Máquina, Servidor, Switch, etc.).
  Caso de Uso: conjunto de secuencias de acciones que ejecuta un sistema y que produce un resultado de interés para un actor del mismo.

  - **Elementos De Comportamiento**

  Interacción: enlaces entre objetos, para solicitar servicios.
  Máquinas de Estados: modela las transiciones y cambios de estado.
  Actividad: secuencia de pasos de un proceso computacional

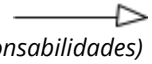
  - **Elementos De Agrupación (Paquetes) y Notación (Notas)**

- **Relaciones**

  1-Dependencia (momentánea)

  2-Asociación
  (permanente, estructural)

  3-Agregación (todo-partes indep)
  La composición es parecida, pero las partes son dependientes.

  6-Generalización 
  (herencia, comparte caracts y responsabilidades)

  7- Realización
  un elemento define un contrato que otro realiza
  (pej: interfaz-objeto)

- **Diagramas:** representación gráfica de un conjunto de elementos. Proyección del sistema.

De estructura:

- *representan relaciones estáticas*, como **Clases** (atrib. y comp. de clases e interfaces), **Paquetes** (organiza paquetes y sus elementos) y **Componentes** (usado en vistas de Diseño).
- *o en tiempo de ejecución*, como **Estructura Compuesta** (muestra la estructura interna de un componente), **Objetos** (*caso especial del de clases*, cuando se quieren ver las instancias de clases) y **Despliegue** (muestra el HW, nodos y artefactos, y la plataforma en la que se corre).

De Comportamiento:

Interacción (Flujo de trabajo, pueden ser: Comunicación, Secuencia, Visión/Interacción, Tiempo), Casos de Uso (interacción entre sistema y entidades externas), Máquina de Estados (Explicado más adelante*), Actividad.**

Máquina de Estados: Diagrama que especifica la secuencia de estados por las que pasa un objeto en su vida, en respuesta a los eventos, y sus respuestas a esos eventos.

Un **estado** tiene un nombre, un evento de E/S, transiciones internas, sub-estados, y eventos diferidos.

Una **transición** es el evento en el cual un objeto cambia de estado ante un evento específico de E/S. Compuesta por un Estado Origen, un Evento de Disparo, la condición de Disparo, un Efecto, y un Evento Destino.

Reglas de UML

- Reglas Semánticas:

De Nombrado (como llamar a los elementos), De Alcance (el contexto que da un significado a un nombre), De Visibilidad (entre los nombres de elementos), De Integridad (el cómo se relacionan apropiadamente los elementos), y De Ejecución (que significa ejecutar un modelo).

- De construcción de Modelos:

Abreviados (Ocultan elementos para simplificar la vista), Incompletos (Faltan algunos elementos, por lo menos al principio del desarrollo), Inconsistentes (no se garantiza la integridad del modelo).

Mecanismos Comunes de UML

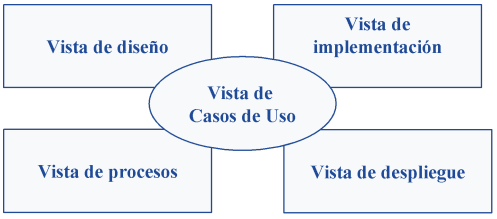
-Especificaciones: es una descripción de la sintaxis y semántica de un bloque de construcción.

-Adornos: elemento usado para realizar indicaciones complementarias. (visibilidad prop clase)

-Divisiones Comunes: las cosas pueden dividirse al menos en un par de maneras (clase-objeto, interfaz-implementación)

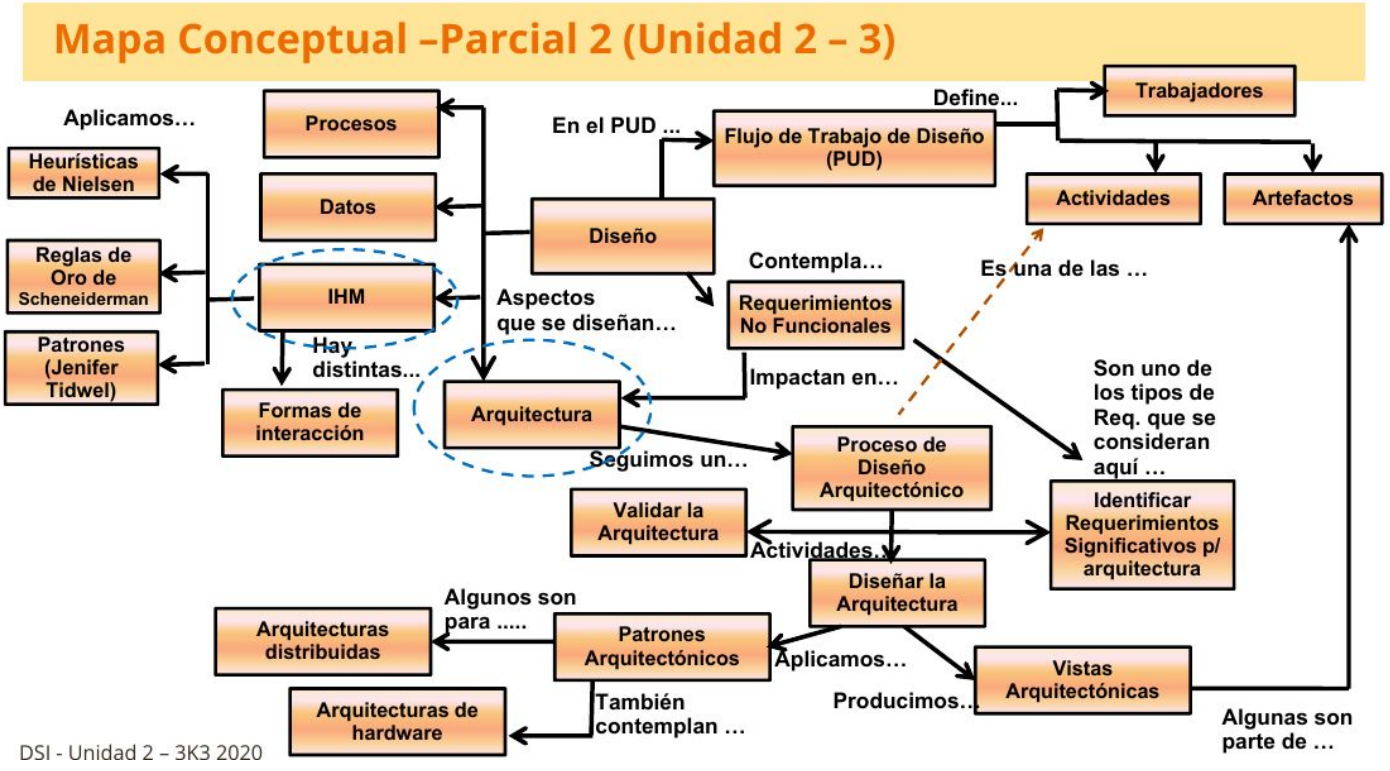
-Mecanismos de Extensibilidad: Como estereotipos, Valores Etiquetados y Restricciones

Vistas de la arquitectura de un Sistema con UML:



Date cuenta de que diagramas se usan para el aspecto estático y dinámico de c/u)
Funcional/Casos de Uso: describen su comportamiento a través de CU**.**
Diseño: dan forma al problema y su solución por medio de clases, interfaces y colaboraciones.
Procesos/Interacción: muestra el flujo de control, las interacciones y los mecanismos de sincronización y concurrencia.
Implementación: componentes y archivos necesarios para poner en ejecución el sistema físico (diagrama artefactos).
Despliegue: nodos de la topología de software sobre la que se ejecuta el sistema

Mapa Conceptual - UNIDADES 2 - 3



WF de Diseño

Mientras que en el Flujo de Análisis (**FA**), la idea era crear el modelo lógico del sistema para capturar la funcionalidad a partir de los requisitos, el Flujo de Diseño **implica elaborar soluciones e implementar estrategias, para lograr un modelo de sistema** (incluyendo su arquitectura)**, que pueda implementarse**, teniendo en cuenta sus requerimientos funcionales y no funcionales.

Es la actividad de modelado principal durante la fase de **Elaboración** y la primera mitad de la de **Construcción**.

En el diseño, se:

- Comprenden en profundidad los **requisitos no funcionales**, y las capacidades y restricciones de las tecnologías, plataformas y lenguajes de desarrollo.
- Se especifica una entrada y **punto de partida para la elaboración**.
- Se **descomponen los procesos de implementación.**
- Se **identifican las interfaces** entre subsistemas.
- **Se crea una abstracción de la implementación del sistema** (o sea que el Sistema es un refinamiento del diseño y viceversa). (Es decir, se construye el Modelo).

Artefactos (7)

(ARQ) Modelo de Diseño: Describe la realización física de los casos de uso centrándose en cómo los requerimientos funcionales y no funcionales, junto a las restricciones, afectan al sistema a implementar. **Es un modelo físico, ya que describe a la Implementación planeada, y específica a la arquitectura y condiciones específicas de desarrollo**, a diferencia del Modelo de Análisis, que es una abstracción de lo que se espera del sistema en términos genéricos. En otras palabras, **el Modelo de Diseño describe la realización física de los Casos de Uso.** Se representa por un **sistema de diseño**, divisible en subsistemas..

(ARQ) Modelo de Despliegue: describe la distribución física del sistema en términos de cómo se distribuye la funcionalidad entre los nodos de cómputo, y las relaciones entre estos últimos mismos.

(ARQ) Descripción de la Arquitectura: Una vista de la arquitectura, es decir, de la organización de un sistema de SW, y de sus artefactos relevantes a tener en cuenta.

- Como vista del Modelo de Diseño, resalta lo más importante del mismo, sus artefactos más relevantes, subsistemas, interfaces y dependencias entre los mismos, las clases de diseño fundamentales y los casos de uso-diseño con funcionalidad crítica.
- Como vista del Modelo de Despliegue, resalta los artefactos relevantes para la arquitectura, como la correspondencia de los componentes sobre los nodos.

(CU) Realización de CU-Diseño: es una **colaboración** en el modelo **de diseño**, que describe cómo se **realiza de manera física un CU**, y se ejecuta en término de sus clases de diseño y objetos en su trazado, gestionando además, los requisitos no-funcionales.

Se especifica mediante Diagramas de Colaboración, Diagramas de Clase-Diseño, Flujos de Sucesos, y los requerimientos de Implementación capturados en el diseño a tener en cuenta *(parecido al WFA, pero más cercano al código, más físico, menos abstracto).*

(COMP) Clase de Diseño: es una abstracción que permite una traducción directa en la implementación del sistema.

- Sus estereotipos, operaciones, parámetros, atributos, etc. son definidos de la misma manera que el lenguaje a usar.
- Al igual que la visibilidad de sus atributos, dependiente del lenguaje a implementar.
- Las responsabilidades se corresponden con los métodos a implementar (*pseudo-código...)*.

(COMP) Subsistema de Diseño: Cumple el papel de contenedor/elemento organización de artefactos de modelo de diseño. Se usan para organizar artefactos del modelo (como clases de análisis, Realizaciones de CU y otros paquetes de análisis) y encapsular elementos del Modelo. Los paquetes deberían ser Altamente Cohesivos, y con un Bajo Acoplamiento. Se pueden anidar paquetes.

(COMP) Interfaz: Define comportamientos (no atributos) y me da flexibilidad para hacer un modelo, proporcionando una "interfaz" común para mandar mensajes, para diferentes implementaciones de responsabilidad. Separa funcionalidad de Implementación.

(COMP) Modelo de Despliegue: Modelo de Objetos que describe la distribución física del sistema en cuanto a la distribución lógica/funcional entre los nodos (HW) del mismo.

Actividades (4)

(1 ARQ) Diseñar la Arquitectura: Se esbozan los Modelos de Diseño y Despliegue, y su arquitectura mediante: los nodos y configuraciones de red, subsistemas y sus interfaces, clases de diseño significativas, y mecanismos de diseño genéricos para la trata de requisitos no funcionales. Se trata de reutilizar, tanto mediante partes de sistemas anteriores, o mediante el uso de patrones.

(2 CU) Diseñar CU: Se diseña un CU para identificar las Clases de Diseño y/o Subsistemas de Diseño necesarios para su funcionamiento, para distribuir el comportamiento y definir los requisitos entre los objetos de diseño participantes, y para capturar los requisitos de implementación.

(3 COMP) Diseñar Clase: Se diseña una Clase contemplando sus atributos, operaciones y métodos, relaciones, dependencias, requisitos de implementación y la implementación de cualquier interfaz requerida.

(3 COMP) Diseñar Subsistema: Se trata de garantizar la Alta Cohesión y Bajo acoplamiento, para la realización de las operaciones tal como estén definidas en las interfaces a proporcionar.

Diferencias entre WFs

CU	ANÁLISIS	DISEÑO
Conceptual (sobre el negocio) y Genérico (vista externa del sistema)	Conceptual (sobre el sistema genérico), Vista Interna sin detalles de implementación	Físico (sobre la implementación del sistema). Vista interna atada a una implementación.
Específica que se necesita	Específica que se debería hacer	Específica que se va a hacer/hizo
Menos Formal	Lenguaje Desarrollador	Re formal, atado a Implementación
Base relativa de coste.	Más caro que CU mas barato que WFD	Más caro todavía
Modela funcionalidad	Bosquejo del diseño	Manifiesto del diseño del sistema
Mantenido siempre	Puede dejar de ser mantenido	Mantenido siempre

Diseño de Software

Es un proceso iterativo por el cual se traducen los requerimientos, en un plano para construir el software, agrupa el conjunto de principios(filosofía general de trabajo), conceptos (definiciones a entender) y prácticas que llevan al desarrollo de un sistema de alta calidad.

Un buen diseño debe implementar todos los requerimientos del modelo de requerimientos, y ajustarse a los requerimientos que desea el cliente, debe ser una guía legible para aquellos que desarrollen o den soporte al sistema, y deberán proporcionar un panorama completo del software, teniendo en cuenta la estructura, funcionalidad y datos desde la perspectiva de implementación.

Cosas a Diseñar:

1. **Arquitectura**: los componentes estructurales del programa y su relación, organizados para cumplir los requerimientos del producto.
2. **Datos**: las estructura de datos necesarias para implementar el software., resulta en el modelo de datos, que se refina e implementa en la Base de Datos.
3. **Procesos**: Definiendo los componentes involucrados en los CU del sistema, y en términos de los procesos manuales de la empresa como resultado de la integración del sistema en la misma.
4. **Interfaces**: la comunicación entre el sistema y su entorno (**IHM**, u otros sistemas) o con sigo mismo (interno)
5. **Formas de Entrada/Salida:** como ingresar y procesar informacion (lotes, línea, o tiempo real).
6. **Procedimientos manuales:** como integrar el software al negocio.

Requerimientos no Funcionales (RNF)

Son aquellos que definen las cualidades o atributos que deben estar presentes en el producto de SW resultante, también se lo suele definir como **consideraciones o restricciones asociadas al sistema.** Presentan la dificultad de que es difícil establecer una "mejor" solución, solo soluciones buenas o malas, ya que normalmente son atributos cualitativos, más que cuantitativos. Sin embargo, siempre se busca que sean testeables y objetivos.

Se clasifican en tres tipos:

- **Técnicos**: restringiendo opciones de diseño para especificar tecnologías que la aplicación debe usar (ej: Java o WXP)
- **De negocio:** restringiendo opciones por razones de negocio, no técnicas. (usar Open Source, porque no hay $$)
- **De Calidad de Producto:** definiendo requerimientos de la aplicación de interés para los usuarios de la aplicación u otros actores (ej: disponibilidad, performance, seguridad, interfaz, portabilidad, etc)

Atributos de Calidad de Producto (ISO 9126 y ISO 25000)

Son parte de los RNF de una aplicación, que capturan muchas facetas sobre cómo los RF son llevados a cabo.

1. **Performance**: en cuanto tiempo debe realizarse un trabajo. Puede ser del proceso o del software en sí.
2. **Escalabilidad**: que tan preparado está el sistema para crecer sin perder calidad o estabilidad.
3. **Modificabilidad**: que tan fácil es realizar cambios en el sistema.
4. **Seguridad**: que tan bien maneja las tareas de autenticación, autorización, encriptación e integridad (alteración msjs)
5. **Disponibilidad**: confiabilidad, que el sistema pueda ser usado cuando se lo necesite.
6. **Integración**: que el sistema pueda ser incorporado a un contexto más amplio.
7. **Portabilidad**: que se pueda usar en diferentes plataformas de HW o SW (dependiente de la tecnología usada).
8. **Testabilidad**: que tan fácil sea de probar.

A tomar en cuenta

- Los requerimientos del cliente y del modelo de análisis
- Que debe ser legible y entendible para quienes implementan o den soporte.
- Que debe ser completo y técnico, rastreable al análisis, y estructurado para admitir cambios.
- Se espera la utilización de patrones de diseño reconocibles.

Arquitectura

Es resultado del conjunto de decisiones significativas que tomamos sobre el producto para cumplir con los requerimientos. Explica cómo se satisfacen los requerimientos funcionales y no funcionales, bajo diferentes puntos de vista.

La arquitectura de un sistema es el artefacto más importante que puede emplearse para manejar estos diferentes puntos de vista y controlar el desarrollo y mantenimiento de un sistema.

Proceso de diseño Arquitectónico:

Consta de 3 actividades:

- **Determinar los requerimientos arquitectónicos:** consiste en identificar los requerimientos que conduzcan al diseño arquitectónico (significativos), y su priorización (alta, media (no primero, pero necesaria), baja (deseable)).

- **Diseño Arquitectónico:** que a partir de los Requerimientos Arquitectónicos, se defina la estructura y las responsabilidades y relaciones de los componentes que constituyan la arquitectura, al elegir un framework y diseñar los componentes de la misma.
- **Validación:** donde se prueba la arquitectura diseñada, ya sea mediante escenarios (casos de prueba) o construyendo prototipos, contra requerimientos existentes o futuros conocidos o posibles.

Diseño Arquitectónico

Como primera etapa/actividad del WFD, el objetivo es **capturar la arquitectura del sistema en componentes y sus relaciones, para definir cómo se organiza el sistema**. Esto se hace al i**dentificar los principales componentes estructurales del sistema y sus relaciones**, y **eligiendo los estilos y patrones de arquitectura a utilizar**, teniendo en cuenta los **requerimientos no funcionales**, para generar como salida un **modelo arquitectónico**, **el cual describa la forma en la que se organiza el sistema como un conjunto de componentes comunicados entre sí.**

Esto se puede hacer a nivel sistema (pequeño) o a nivel empresarial(grande), teniendo en cuenta la interacción entre varios sistemas, programas y componentes de programa.

No existe una solución perfecta, pero se busca la solución óptima, en base a las circunstancias. Se debe considerar el volumen de datos, la funcionalidad más demandada del negocio, la distribución geográfica del negocio, la distribución de la infraestructura, y la forma de guardado de los datos.

Como salidas del diseño arquitectónico, se espera la definición de la distribución geográfica de los requerimientos computacionales, la forma en que estos estén configurados, la capacidad computacional requerida tanto a nivel cliente como a nivel servidor, los mecanismos y formas en que los recursos computacionales se comuniquen, el sistema operativo a utilizar o soportados, el paradigma de desarrollo a utilizar, y las tecnologías a utilizar para el desarrollo (lenguajes, DB).

El **rol del arquitecto** en el diseño arquitectónico consiste en revisar y negociar los requerimientos, diseñar la arquitectura, documentarla, comunicarla a las partes interesadas, configurar la arquitectura de hardware, y ayudar con la planificación y calendarización de tareas.

La **documentación de la arquitectura tiene varios propósitos**, entre ellos servir como base para la comunicación, como herramienta de análisis y revisión, y para su reutilización a posteriori.

Para la documentación, se hace uso de 4 vistas arquitectónicas, que se describen más adelante.

- Elección del modelo Arquitectónico (Framework):

Aunque cada sistema es único, en general se suelen utilizar arquitecturas similares que reflejen los requerimientos fundamentales del dominio, por lo que al diseñar un sistema, se debe decidir qué tanto tienen en común estas arquitecturas comunes con el sistema, para determinar qué tanto se puede reutilizar de las primeras en este último. La arquitectura de un sistema, entonces, puede (y suele) basarse en uno o varios **patrón arquitectónico** particular, que suele ser elegido normalmente en base a los RNF del sistema, normalmente los de tipo:

- **Desempeño**: haciendo que operaciones críticas se realicen con la menor comunicación posible para optimizar la respuesta.
- **Seguridad**: sugiriendo usar capas para aislar y proteger datos o información.
- **Protección**: sugiriendo centralizar la protección de un sistema en pocas capas, para evitar costos y problemas de validación.
- **Disponibilidad**: estableciendo la necesidad de tener redundancia donde sea posible, para asegurar el acceso y funcionamiento del sistema.
- **Mantenibilidad**: estableciendo la necesidad de que el sistema esté conformado por componentes autocontenidos con una gran independencia.

Sin embargo, al modelar tenemos que tener en cuenta de que al centralizar operaciones, aumentamos la performance, pero disminuímos la mantenibilidad, si introducimos redundancia, aumentamos la disponibilidad, pero la seguridad se vuelve más difícil, y si centralizamos la protección, la mantenibilidad se ve perjudicada.

- Distribución de componentes:

La idea aquí es identificar los componentes de la aplicación, sus interfaces y responsabilidades, y luego distribuirlos entre los nodos de red.

Vistas Arquitectonicas:

Representaciones del sistema bajo una perspectiva, que muestra solo la información que se necesite/quiera ver, y se oculta el resto. En PUD se hace uso de 4 vistas:

- **la Vista Lógica**: que describe los elementos significativos arquitectónicamente y sus relaciones. Captura básicamente la estructura de la aplicación con diagramas de clases o equivalentes.
- **La vista de Procesos**, que se enfoca en describir la concurrencia y elementos de comunicación de una arquitectura (sincrona y asincrona).
- **La Vista Física o de Implementación**, que describe cómo se mapean los principales procesos y componentes en nodos de hardware,
- **y la vista de desarrollo**, que captura la organización interna de los componentes de software en paquetes de clase.
- Estas vistas se unen por los casos de uso arquitectónicamente significativos (la vista de Casos de uso o Funcional).

Bajo el método **Siemens Four-Views** en cambio, las vistas son:

- **Conceptual**: donde se relaciona a la funcionalidad con sus componentes
- **Módulos**: donde se describe como el sistema se descompone en subsistemas y módulos.
- **Ejecución**: que describe la estructura del sistema en términos de los objetos que participan en tiempo de ejecución ante ciertos escenarios de funcionalidad.
- **Código**: que se ocupa de la organización de los artefactos de software, en términos de su código fuente, y el versionado del mismo.

Patrones Arquitectonicos

Una patrón es una descripción abstracta estilizada de una buena práctica, que se ensayó y puso a prueba en diferentes sistemas y entornos. Un patrón arquitectónico debe describir una organización de sistema que ha tenido éxito en sistemas previos. Debe incluir sobre cuándo es adecuado usar dicho patrón, así como las fortalezas y debilidades del mismo.

Existen patrones en 3 niveles de solución: bajo nivel y detallados (lenguajes de programación), medio (objetos y clases comunes), y de alto nivel, o arquitectónicos (componentes y módulos).

Cabe resaltar la diferencia entre patrón y estilo arquitectónico: un sistema normalmente tiene un único estilo, mientras que los patrones están en una menor escala, y puede haber varios en ese sistema.

Otra diferencia está en el estado del patrón, si es embebido el patrón está aplicado, mientras que cuando es platónico es básicamente la forma idealizada del patrón, la que se ve en libros.

Dentro de los patrones Arquitectónicos, encontramos 9 para resaltar:

1. **Layered**: donde se organiza el sistema en capas con funcionalidad relacionada solo con la capa continua, con Interfaces claras y bien definidas.
2. **Repository**: consiste en un conjunto de componentes independientes, que se relacionan sólo por medio de un modelo central.
3. **MVC (Modelo Vista Controlador):** Separa la presentación e interacción de los datos del sistema, al separarlo en tres componentes que interactúan entre sí, El modelo maneja los datos del sistema y sus operaciones relacionadas. La vista define y gestiona cómo se presentarán los datos al usuario. El controlador dirige la interacción con el usuario y comunica al (los) modelo(s) con la vista.
4. **Publish-Subscribe:** Consiste en componentes que publican eventos y otros que se suscriben a ellos mediante un tópico. Los suscriptores pueden suscribirse o desuscribirse, y los publicantes solo necesitan publicar un evento, el tópico se encarga de la entrega de los eventos a los suscriptores.
5. **Messaging:** Se hace uso de una cola, para desacoplar a un cliente de un servidor. El cliente puede enviar un mensaje por medio de la cola, con la seguridad de que este será eventualmente entregado, aunque la red se caiga, el cliente pierda su conexión después de hacer el envío, o el receptor no esté disponible.
6. **Pipe & Filter:** Los datos fluyen de un componente a otro para su procesamiento a medida que ingresan, el procesamiento de datos se organiza de forma que cada componente de procesamiento sea discreto (solo dependa de la entrada) y realice un tipo de transformación de datos. Se suele usar en aplicaciones de procesamiento de datos complejos.
7. **Batch-Sequential:** Los datos fluyen de una capa a otra y son procesados incrementalmente, a diferencia de Pipe & Filter, cada etapa completa todo su procesamiento antes de escribir su salida. El resultado puede fluir entre capas o almacenarse en un recurso persistente externo.

8. **Broker**: Existe un concentrador de mensajes, que toma mensajes en ciertos puertos de entrada, para procesarlos y rutearlos a receptores en ciertos puertos de salida.
9. **Process Coordinator:** Parecido a repository, se basa en un coordinador que encapsula la secuencia de pasos para completar un proceso de negocio, y dirige a los servidores que realizan las transformaciones en el orden definido por el proceso, para luego emitir el resultado.

Arquitectura de sistemas Distribuidos

Un sistema distribuido es aquel que implica numerosas computadoras, en contraste con los sistemas centralizados en que todos los componentes están en la misma computadora. El sistema debería ser transparente para los usuarios, es decir, que los usuarios no deberían notar que en realidad están interactuando con un grupo de computadoras.

La idea en general de distribuir un sistema, se basa generalmente en la **escalabilidad** y **estabilidad** del mismo, esto se hace posible por medio de la **compartición de recursos** a través de una red, la apertura y uso de **estándares** para comunicarse, y la **concurrencia** de procesos que esto termina posibilitando. Además, la disponibilidad de muchas computadoras permite tener redundancia, y aumentar la **tolerancia a fallos**.

La **complejidad** añadida, sin embargo, es un punto a considerar, ya que dificulta la tarea de administrar la **seguridad**, **probar** el sistema, y administrarlo, y añade más elementos que pueden fallar al sistema, como la **red** sobre la que se comunican los nodos. Por esto, se suele utilizar **Middleware**, que consiste básicamente en software enlatado que permite gestionar y ayudar a la comunicación entre las partes del sistema.

Los recursos computacionales pueden **distribuirse** de forma espejada, en rack, o como granja de servidores.

Patrones Arquitectónicos para sistemas Distribuidos

Se busca equilibrar rendimiento, confiabilidad, seguridad y manejabilidad del sistema. Mencionamos 5 alternativas:

Maestro-Esclavo: en el que el proceso maestro es el responsable de la computación, coordinación, comunicación y control de los procesos esclavos, mientras que estos últimos se dedican a acciones específicas, como adquisición de datos de sensores, o el procesamiento de tareas específicas simples. Se suele usar en sistemas de tiempo real.

Cliente-Servidor: se modela como un conjunto de servicios que son provistos por clientes, y un conjunto de clientes que consumen esos servicios, los clientes deben conocer a los servidores, pero los servidores no necesitan conocer a los clientes. Puede realizarse con **dos capas**, o **múltiples**, esto permite mayor flexibilidad y escalabilidad, a costa de mayor complejidad.

N-Tier: describe la separación de la funcionalidad en diferentes segmentos, de manera similar a la arquitectura por capas, pero cada segmento es un nivel que se encuentra físicamente en un equipo independiente.

Componentes distribuidos: se diseña el sistema como un conjunto de servicios sin asignarlos a capas, los nodos consumen estos servicios a través del middleware. Esto aumenta mucho la complejidad y la dependencia al middleware, al mismo tiempo que permite flexibilidad a la hora de agregar nuevos servicios.

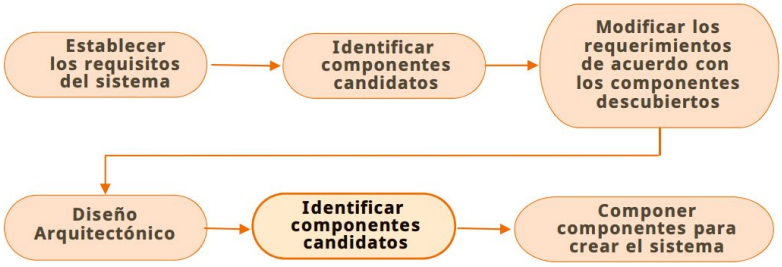
P2P: no se hacen distinciones entre clientes y servidores, todo cliente puede ser servidor. Permite sacar ventaja del poder computacional de los nodos de una red de computadoras potencialmente enorme. Exige coordinación para evitar doble procesamiento, pero puede escalar infinitamente.

Map-Reduce: se basa en una serie de nodos (mappers) que procesan y transforman los datos de entrada de forma paralela, y un nodo final (reducer) que combina los resultados de los nodos para producir el resultado final y almacenarlo.

Ingeniería de Software Basada en Componentes:

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

Así, la ISBC consiste en definir, implementar e integrar o componer los componentes independientes e imprecisos en los sistemas. Así, podemos reutilizar piezas de código preelaborado que permiten realizar diversas tareas, conllevando a diversos beneficios como las mejoras a la calidad, la reducción del ciclo de desarrollo y un mayor ROI. La idea es usar componentes reutilizables, en vez de re-implementarlos una y otra vez.



Proceso ISBC

Para esto se basa en una serie de **fundamentos**:

- Construcción y uso de componentes independientes definidos/especificados por completo por sus interfaces.
- Estándares de componentes que faciliten la integración de los mismos con otros.
- Utilizar un middleware que permita y facilite la integración de componentes.
- Un proceso de desarrollo afín a la ISBC (suficientemente flexible, dependiendo de la funcionalidad de los componentes disponibles).

Así mismo, presenta una serie de **problemas**:

- **Confiabilidad de los componentes:** los componentes al fin y al cabo, son piezas externas de SW, por lo que pueden tener problemas, ser maliciosos, o no estar bien documentados.
- **Certificación de Componentes:** los componentes al ser posiblemente de un tercero, pueden no estar certificados, y quien paga la certificación?.
- **Propiedades no deseadas:** los componentes pueden tener comportamientos no deseados para el sistema objetivo, y en estos casos se debe “trabajar alrededor de los mismos”
- **Equilibrio de Requisitos:** debemos hacer un equilibrio entre los requerimientos del sistema, y los componentes disponibles.

Composición de Componentes:

Proceso de integrar componentes uno con otro, usando “código pegamento” para crear un sistema u otro componente.

- **Composición secuencial**: se usa código pegamento para llamar a los componentes en el orden correcto (asegurando que el resultado de un componente sea compatible con la entrada esperada por el siguiente).
- **Composición Jerárquica:** un componente llama directamente a los servicios que otro componente ofrece, no es necesario código pegamento.
- **Composición aditiva**: se combinan dos o más componentes en un nuevo componente, que utiliza código pegamento para proporcionar una interfaz de componente combinada y brindar algún servicio.

Diseño Interacción Humano- Máquina

Asegura la funcionalidad y usabilidad del sistema, proporcionando un soporte efectivo a las interacciones con el usuario, y posibilitando una experiencia placentera de uso. La idea es brindarle la mayor utilidad posible al usuario con una interfaz que sea fácil de usar y aprender, y ser productivo.

La **usabilidad** es el conjunto de atributos que permiten a los diseñadore medir sus sistemas e interfaces en términos de la experiencia de los usuarios que las utilizan, atributos como el aprendizaje (feedback), la velocidad de funcionamiento(productividad), la robustez (tolerancia a errores del usuario), la recuperación(ante errores del usuario) y la adaptación (capacidad de adaptarse a diferentes situaciones del usuario).

Se tienen diferentes **formas de interacción**, como la **manipulación directa** (videojuegos), la **selección de menús**, el **rellenado de formularios** (empresarial), o el uso **de lenguaje de comandos**, o de **lenguaje natural**.

Se suele hacer uso de prototipos, y pruebas con usuarios potenciales, para comprender las circunstancias y los cursos de acción que los usuarios esperan tomar, para finalmente definir los elementos de interfaz que le permitan a los usuarios realizar sus tareas.

Para el diseño de interfaces, se pueden mencionar 3 recursos/guías:

- **Heuristicas de Nielsen:** una serie de principios generales de diseño, a seguir para facilitar el uso del sistema. Como por ejemplo *"Visibilidad del estado del sistema", "Correspondencia entre el sistema y el mundo real", y "Consistencia y estándares", y "Prevención de errores"*.
- **Reglas de Oro de Schneiderman:** parecido al anterior, con reglas parecidas.
- **Patrones de Diseño de Tidwell**

Prototipado de Software

Un prototipo es una versión inicial de un sistema de software que se utiliza para demostrar conceptos, probar opciones de diseño y, en general, informarse más del problema y sus posibles soluciones.

Puede usarse para validar requerimientos, explorar soluciones de software o apoyar al diseño de interfaces de usuario durante la etapa de diseño, o para realizar pruebas con el sistema final a entregar al cliente.

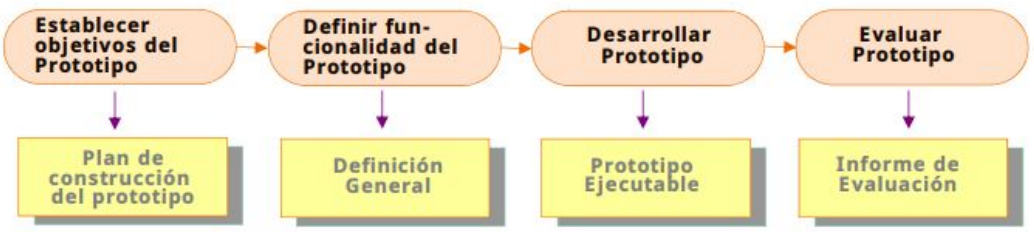
Se los suele usar cuando el dominio es riesgoso (no se conoce o comprende, o es muy complejo), cuando se usan nuevos métodos o tecnologías, cuando se pretende evaluar la viabilidad o el impacto del sistema en la organización, o cuando el coste por rechazo de parte de los usuarios en caso de que el sistema no sea correcto sea muy alto.

Los prototipos pueden ser **rápidos/desechables** (sirve para su tarea, y luego se desecha, se permiten y esperan diferencias con el sistema final) o **evolutivos**, que comienzan con un sistema relativamente simple que implementa los requerimientos más importantes o mejor conocidos y se lo amplía y mejora hasta convertirse en el sistema final.

Características de los prototipos

- Aplicación que funciona
- Posee funcionalidad limitada
- Incorpora sólo algunas características del sistema final
- Poca fiabilidad
- Insumen poco presupuesto (relativamente) y tiempo de desarrollo

Proceso de Desarrollo de Prototipos



Establecer los objetivos de un prototipo es muy importante, para manejar las expectativas, así como lo es **definir la funcionalidad** esperada del mismo, para centrarse en lo que el prototipo espera validar.

Conceptos del Diseño Orientado a Objetos

Encapsulamiento

Es el proceso de almacenar en una misma sección los elementos de una abstracción que constituyen su estructura y su comportamiento; sirve para separar el interfaz contractual de una abstracción y su implantación. Si hacemos que la única forma de acceder o cambiar los datos internos de un objeto sea por medio de las operaciones que expone, decimos que el estado interno está encapsulado.

Herencia

Es el mecanismo por el cual una clase nueva se crea a partir de una clase existente, definida en tiempo de compilación. La herencia proviene del hecho de que la subclase (la nueva clase creada) contiene los atributos y métodos de la clase primaria. Es la mayor forma de acoplamiento posible entre clases, y su encapsulamiento es débil, ya que cambios en la clase madre provoca cambios en las subclases y las interioridades de las clases primarias usualmente son visibles para las clases hijas.

Esto nos otorga también una propiedad que se llama polimorfismo, que quiere decir que cualquiera de las formas hijas pueden adoptar la forma primaria, o lo que es lo mismo, la primaria puede estar enmascarando a cualquier hijo mediante su forma.

Composición

Composición quiere decir que tenemos una instancia de una clase que contiene instancias de otras clases que implementan las funciones deseadas. Con composición, la funcionalidad de un objeto se obtiene ensamblando objetos en tiempo de ejecución para obtener una funcionalidad más compleja.
La composición requiere que los objetos participantes tengan interfaces bien definidas, pero mejora el encapsulamiento, ya que el/los objetos participantes son solo accesibles por medio de su interfaz, y las clases no se ven tan acopladas como con la herencia, lo que las vuelve más simples de entender y testear.

Delegación

Por medio de la delegación, logramos que la composición sea tan potente para la reutilización como la herencia, al hacer que el objeto compuesto delegue el tratamiento de una petición a su delegado.

Clase Abstracta

Se le dice a las clases que tiene algunos (o todos) sus métodos sin implementar. Estos métodos *deben* ser implementados por las clases hijas. Las clases abstractas no se pueden implementar.

Interfaz

Cada operación declarada en una clase u objeto, tiene su **firma**. Una firma está formada por un nombre, parámetros y tipo de valor de retorno. A una sumatoria de firmas, se le denomina Interfaz.

Una **interfaz** especifica un conjunto de características públicas. **La idea es separar la especificación de funcionalidad de su implementación en código.** Una interfaz no es instanciable, es un contrato que puede ser realizable por una o más clases.

Sin embargo, cuando un objeto cumple con una interfaz X, se le denomina objeto de **tipo** X. Las interfaces son combinables y el contrato que forman puede ser cumplido total o parcialmente. El tipo de un objeto sólo se refiere a las operaciones que puede realizar, no a su implementación (por lo que puede ser de cualquiera e incluso más de una clases)

Herencia de Clases vs Herencia de Interfaces

La herencia de clases (generalización) define la implementación de un objeto, en términos de la implementación de otro. La herencia de tipos (realización) define cuando se puede usar un objeto en vez de otro.

Framework

Es un conjunto de clases concretas que determinan una arquitectura, definiendo la estructura de una aplicación, constituyendo un diseño reutilizable. Más grandes, especializados y concretos que patrones.

Principios del Diseño Orientado a Objetos

Programar para Interfaces, no para una implementación

Si programamos en base a contratos entre objetos, de que es lo que se espera de cada operación, de esta manera, los objetos no necesitan la clase o implementación del objeto con el que interactúan, solo necesitan saber que el objeto con el que interactúan es del mismo tipo. Esto tiene enormes ventajas en términos de modificabilidad del código.

Principios SOLID

Conjunto de principios y buenas prácticas que se emplean en el diseño y la programación orientada a objetos, que permiten escribir software de calidad, legible, reusable, escalable, entendible y testable.

Single Responsibility Principle

Una clase debería concentrarse en solo hacer una cosa, para que si cambia el requisito de esa cosa, solo esa clase debe ser modificada.

Open/Closed Principle

Deben ser abiertas para la extensión, ya sea por herencia, delegación o composición, pero cerradas para la modificación. *Esto quiere decir que deberíamos preparar las clases para que su comportamiento sea extensible por herencia o composición.*

Liskov Substitution Principle

Las funciones que usen punteros o referencias a clases base, deberían ser capaces de usar objetos de clases derivadas de las base sin saberlo. *Es decir, no romper la interfaz de la clase base al heredar.*

Interface Segregation Principle

Los clientes deberían ser forzados a depender de interfaces que no utilicen. *Es decir, que al hacer clases que implementan interfaces, las interfaces se usan completamente , caso contrario es mejor descomponer y tener interfaces más pequeñas.*

Dependency Inversion Principle

Las clases de alto nivel no deberían depender de clases de bajo nivel. Ambas deberían depender de abstracciones. Las abstracciones no deberían depender de los detalles, los detalles deberían depender de las abstracciones. *Hagamos que el código dependa de abstracciones y no concreciones, usando muchas interfaces y clases abstractas siempre que se pueda, y exponiendo, por constructor o parámetros, las dependencias que una clase pueda tener.*

Patrones de Diseño

Si queremos que nuestro diseño de SW orientado a objetos sea reutilizable y efectivo, no deberíamos resolver cada problema partiendo de cero, sino que deberíamos reutilizar soluciones que ya fueron probadas en el pasado. Cuando se encuentra una solución buena, se reutiliza una y otra vez.

Los patrones resuelven estos problemas de diseño, y hacen que los diseños sean más flexibles, elegantes y reutilizables. Cada patrón **nomina**, **explica** y **evalúa un diseño importante** y recurrente en los sistemas, con el objetivo de representar esa experiencia de diseño de forma de que pueda ser **reutilizada** por otras personas.

Usar patrones nos permite entonces, **reutilizar buenos diseños**, **elegir alternativas** que hagan a un sistema reutilizable, y lograr un diseño de sistema más **rápidamente**, y finalmente, nos ayudan a la hora de **documentar y mantener el sistema**, ya que al ser comunes, facilitan el entendimiento de quienes suelen tener que leer la documentación.

Cada patrón está **formado** por:

- **Nombre**: que describe con una o dos palabras el problema y su solución
- **Problema**: describiendo cuándo aplicar el patrón. Explica el problema y su contexto.
- **Solución**: describe de forma abstracta los elementos del diseño, su estructura y colaboraciones.
- **Consecuencias**: son los resultados, así como las ventajas y desventajas de aplicarlo.

Usos de patrones:

- **Encontrar los objetos apropiados:** descomponer un sistema en objetos suele ser una de las tareas del Diseño Orientado a Objetos. Los patrones nos ayudan a identificar abstracciones menos obvias.
- **Determinar la granularidad de los objetos:** ¿que debería ser un objeto?
- **Especificar** las **interfaces** de los objetos (sus responsabilidades)
- **Especificar** las **implementaciones** (por medio de su clases)
- **Favorecer la reutilización** de código, por medio de herencia, composición o delegación
- **Diseñar para el cambio**: anticipando nuevos requerimientos y cambios.

Aplicarlos implica, antes que nada, conocerlos, conocer y comprender cómo funcionan, qué problema solucionan, y las consecuencias de su aplicación, elegir uno o más de uno para aplicar en nuestra estructura, para luego identificar cómo aplicar cada solución en el contexto de diseño particular.

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

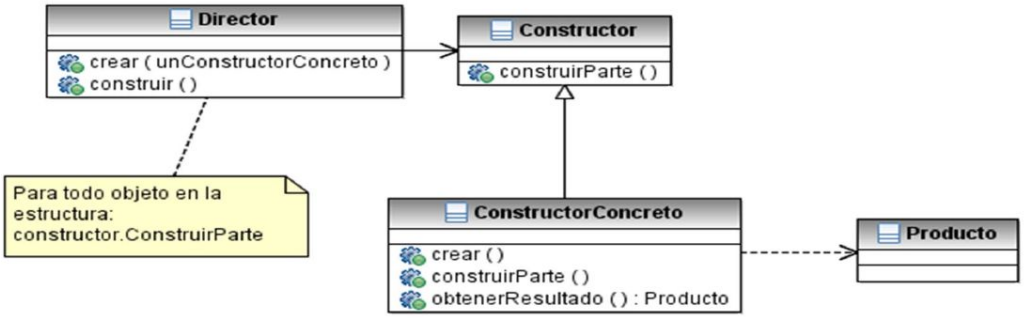
Patrones de Creación

Builder

Descripción: separa la construcción de un objeto complejo, del objeto en sí, de forma que el mismo proceso pueda crear diferentes objetos. (Fabricación Pura)

Uso: Cuando el proceso de construcción del objeto es compleja y hay más de un objeto construible.

Estructura:



Colaboración:

- El cliente crea el director, pasando un *ConstructorConcreto* por parámetro.
- El director va llamando a cada una de las funciones (que conoce de la interfaz *Constructor*) del constructor cuando el cliente llama a *construir*().
- El cliente llama a *obtenerTodo*() al finalizar para obtener el producto.

Consecuencias:

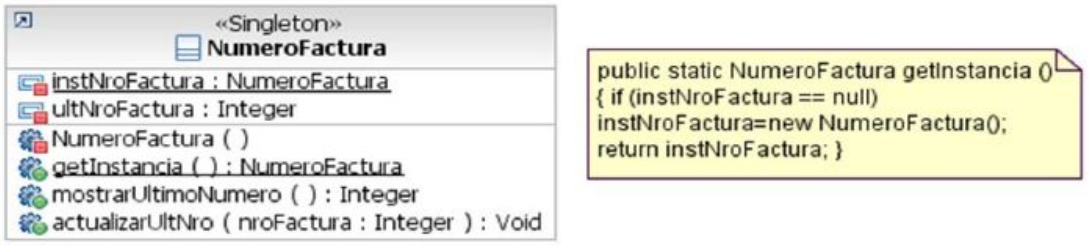
- Permite construir objetos diferentes con el mismo cliente, sin variar otra cosa que nuevos *ConstructoresConcretos*
- Permite extraer la construcción de objetos complejos en varios pasos del objeto.

Singleton

Descripción: garantiza la uniquisidad de una clase, con un punto de acceso global a la misma.
Evita el uso de variables globales, que ensucian el espacio de nombres y son más complicadas de mantener y asegurar. La clase se asegura de su propia instancia por medio de métodos estáticos.

Uso: cuando se tiene una instancia única de clase, o se tienen variables globales que se deseen reemplazar.

Estructura:



Colaboración: El singleton define una operación *getInstancia*() que permite que los clientes accedan a su única instancia. A la hora de interactuar cada cliente debe llamar primero a esta operación.

Consecuencias:

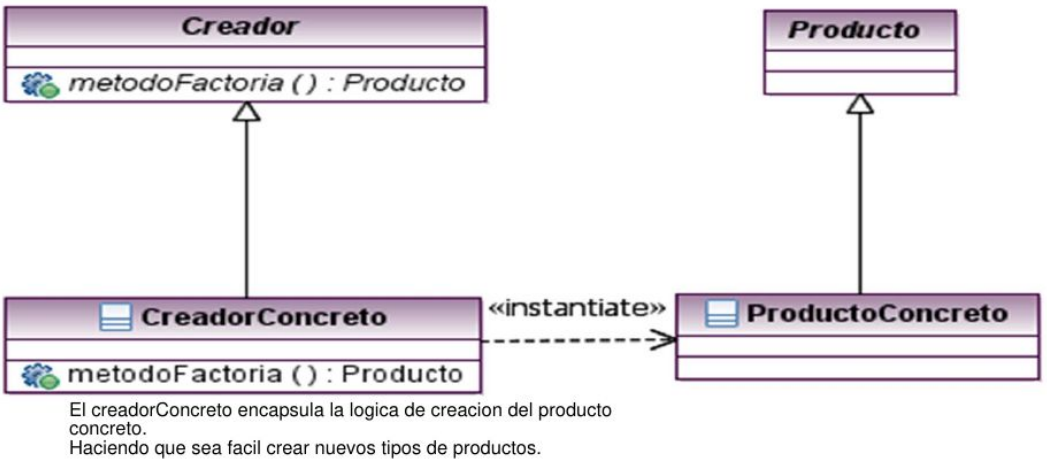
- Permite controlar el acceso y modificación de información de la clase.
- Mantiene el espacio de nombres limpio
- Se puede cambiar de enfoque y hacer más de una.

Factory Method

Descripción: define una interfaz para crear un objeto, pero sólo las subclases decidan qué clase de objeto instanciar. Así, la clase delega en las subclases la creación de objetos. (Fabricación Pura)

Uso: cuando la clase primaria no prevé que producto va a tener que crear.

Estructura:



Colaboración: El *Cliente* se apoya en subclases de *Creador* para que este devuelva la instancia de *Producto* correcta. (opcional: la clase madre *Creador* puede construir un *Producto* default). Se crea el *CreadorConcreto* y se envía al objeto Cliente para que este llame a *metodoFactoría*().

Consecuencias:

- Facilita extender la creación de subclases de Producto.
- Puede suceder que los CreadorConcreto sólo tengan la responsabilidad de crear productos (mal).

Patrones Estructurales

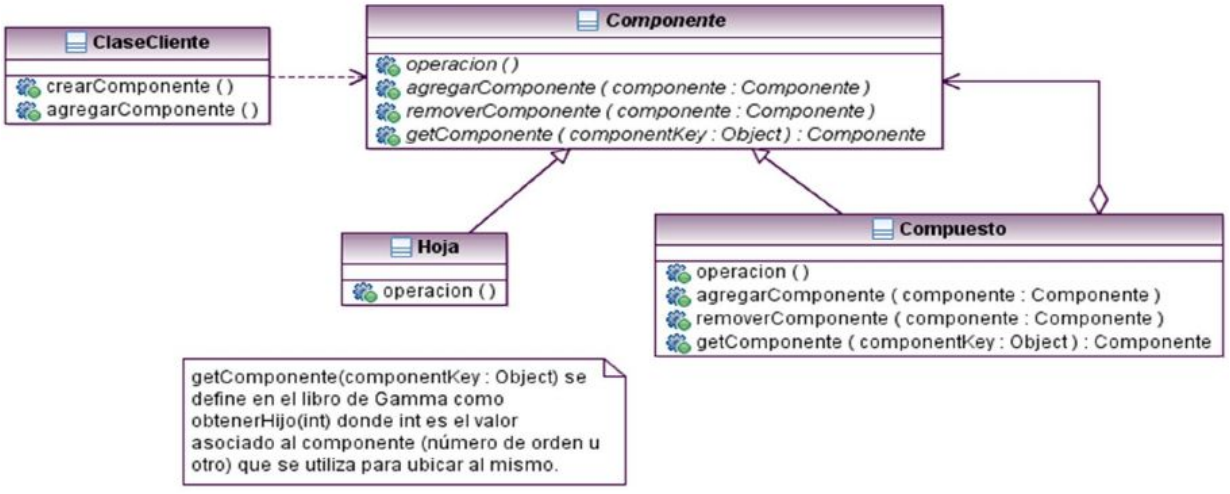
Composite

Descripción: compone objetos en estructura de árbol para representar jerarquías que actúen de igual manera tanto para hojas, como para ramas (compuestos), mientras se mantiene una jerarquía parte-todo.

El patrón describe cómo usar composición recursiva para que los clientes no sean quienes tengan que hacer esa distinción, usando como base una clase abstracta que represente tanto a las primitivas, como a los compuestos.

Uso: Se aplica para jerarquías parte-todo, y para cuando se desea obviar diferencias entre composiciones y objetos individuales.

Estructura:



- **Componente:** define la interfaz común para todos los nodos, e incluye operaciones para acceder a hijos y padres.
- **Hoja:** representa a nodos sin hijos (hoja), y los comportamientos primitivos.
- **Compuesto:** representa nodos con hijos, y almacena otros nodos (hijos o compuestos), sus operaciones son delegadas a los nodos hijos.
- **Cliente:** trata a todos los componentes por medio de la interfaz Componente

Colaboración:

- Al tratar con una hoja, se realizan las operaciones primitivas.
- Al tratar con un compuesto, se realizan las operaciones primitivas de sus hojas.

Consecuencias:

- Se simplifica la interfaz con el cliente en estructuras complejas.
- Facilita generar estructuras complejas.

Decorador

Descripción: asigna responsabilidades a un objeto de forma dinámica en tiempo de ejecución, como alternativa a subclases.

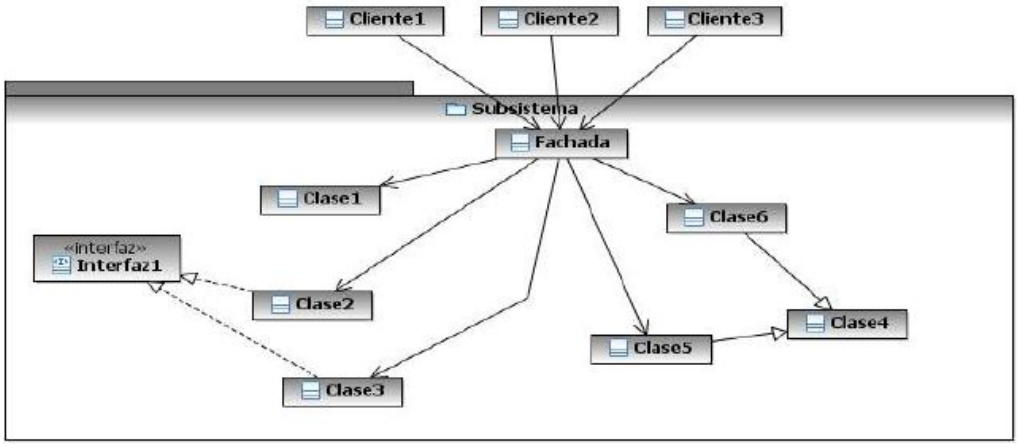
Colaboración: Lo hace implementando la interfaz del objeto decorado (ruteando las peticiones al mismo) con un puntero al mismo, y aplicando lógica extra antes o después de redirigir las llamadas al mismo.

Fachada:

Descripción: proporciona una interfaz unificada/simple para un conjunto de interfaces de sistema.

Uso: la idea es reducir las dependencias entre paquetes o subsistemas, al ofrecer una interfaz simplificada.

Estructura:



Consecuencias:

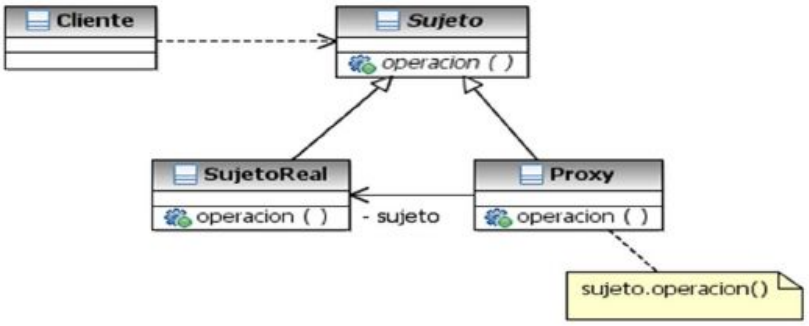
- Facilita el acceso/uso de un subsistemas a sus clientes, al simplificar su interfaz.
- Proporciona un acoplamiento débil del subsistema con sus clientes, mientras se mantenga la interfaz se pueden reemplazar los componentes internos.

Proxy

Descripción: proporcionar un sustituto a un objeto, para controlar el acceso/creación a/de dicho objeto. Usa delegacion.

Uso: diferir el costo de crear un objeto hasta que sea necesario usarlo: creación bajo demanda, para acceso remoto (cuando el sujeto está en otro espacio de direcciones), para controlar el acceso al sujeto.

Estructura:



Consecuencias:

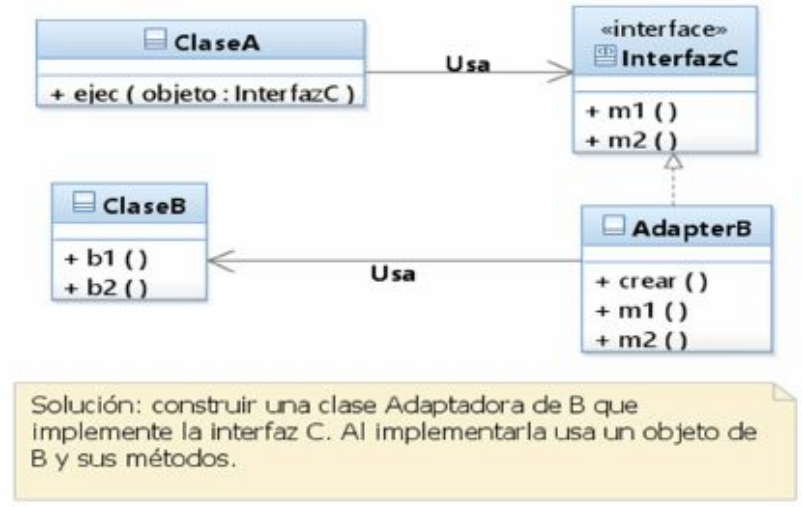
- Introduce un nivel de indirección al acceder a un objeto. La indirección puede ser para:
  - Proporcionar acceso a objetos de diferente espacio de direcciones
  - Optimizar el acceso, al crear o copiar datos bajo demanda
  - Controlar el acceso a los objetos.

Adapter

Descripción: define una interfaz de una clase en otra distinta, que es la que esperan usar los clientes. Permite que cooperen clases que de otra forma no serían compatibles. Puede usarse delegación (objeto) o herencia múltiple (clases).

Uso: se desea usar una clase existente y su interfaz no concuerda con la que se necesita.

Estructura:



Consecuencias:

Si se adapta a una clase y sus subclases (adaptador de objeto), se permite adaptar una clase y todas sus subclases, pudiendo añadir funcionalidad a todos los adaptados a la vez. Pero modificar alguna de las subclases se vuelve más complicado (a menos que se mantenga la interfaz).

Si se adapta a una clase específica por medio de herencia, el adaptador puede modificar al adaptado, al ser una subclase de alguna clase concreta de Adaptado.

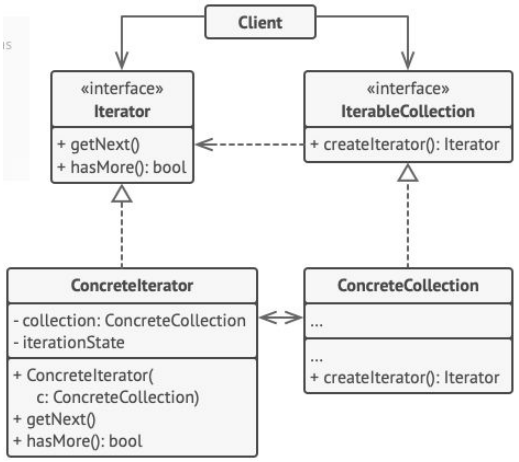
Patrones de Comportamiento

Describen no sólo patrones de clases y objetos, sino también patrones de comunicación entre ellos, producto de flujos de control.

Iterador

Descripción: proporciona un modo de acceder secuencialmente (o incluso en simultáneo) a los elementos de un objeto, sin exponer su implementación o estructura interna.

Estructura:



Colaboración: El *Iterador* es el encargado de recorrer la *Lista*, bajo la lógica que desee, permitiendo así variaciones de recorrido más complejas, sin exponer la lógica interna de la *Lista*

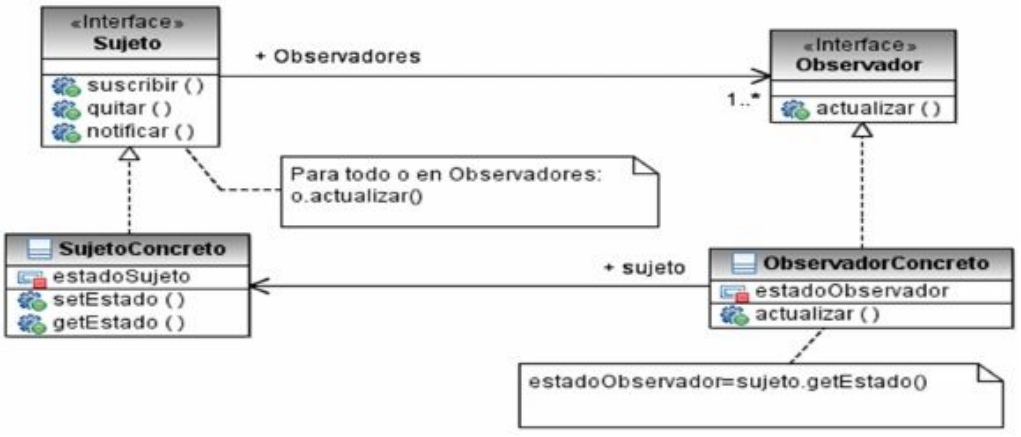
Observer

Descripción: define una dependencia uno-muchos entre objetos, por medio de la cual un objeto al cambiar de estado notifica y actualiza el estado de todas sus dependencias.

Un *Sujeto* puede tener 0..* *Observadores*, el sujeto al cambiar de estado notifica a los *observadores*, y estos consultan al *sujeto* y actualizan su estado. Una variante es hacer que la notificación contenga el nuevo estado.

Uso: cuando hay interdependencia entre objetos, pero se desea tratarlos de forma independiente, cuando un objeto depende de otros, pero no queremos acoplarlos demasiado.

Estructura:



Consecuencias:

- Permite la difusión de eventos y mensajes.
- Proporciona una comunicación basada en eventos muy flexible
- Se debe considerar como escala. Si es mejor hacer Pull o Push.
- Puede generar actualizaciones cíclicas.

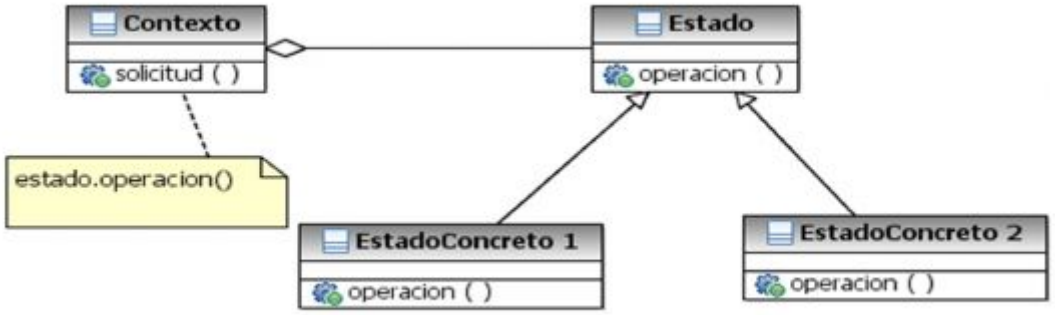
State

Descripción: permite que un objeto cambie su comportamiento cada vez que cambia su estado interno.

Se crean clases "Estado" para cada estado posible, que implementan una interfaz común. En lugar de definir en el objeto de contexto las operaciones que dependen del estado, se delegan en el objeto actual.

Uso: el comportamiento de un objeto depende de su estado y sus métodos contienen lógica condicional (*switch*) dependiente del estado, acoplando al contexto el comportamiento que es dependiente del estado. Se desea tener mayor flexibilidad a la hora de agregar nuevos estados y comportamientos dependientes de los mismos.

Estructura:



Se aplica mediante herencia (subclases de *Estado*, c/u implementando la interfaz *Estado* de forma diferente). El *Contexto* mantiene una instancia de subclase (el *EstadoConcreto* actual) y en sus peticiones se envía a sí misma como parámetro para posibilitar el cambio de estado en runtime.

Consecuencias:

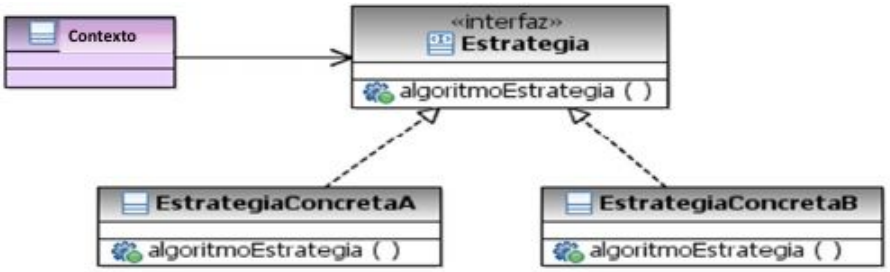
- Coloca el comportamiento asociado a un estado, en su clase dedicada.
- Las transiciones de estado son más explícitas.
- Flexibiliza la creación de nuevos estados o cambios de lógica de transición.
- Los estados pueden ser singleton.

Strategy

Descripción: define una familia de algoritmos que poseen la misma interfaz, encapsula su implementación para hacerlos intercambiables e independientes respecto al cliente que los use. De esta manera, permite implementar variantes de algoritmos de forma prolija y encapsulada, cambiando un switch gigante por clases Estrategia, y haciendo al contexto configurable, por mantener una referencia a la *EstrategiaConcreta* a utilizar.

Uso: se desea configurar una clase con varios comportamientos posibles, o se necesitan diferentes variantes de un algoritmo complejo. O una clase tiene sentencias switch gigantes y se desea simplificar el alcance de la clase.

Estructura:



Consecuencias:

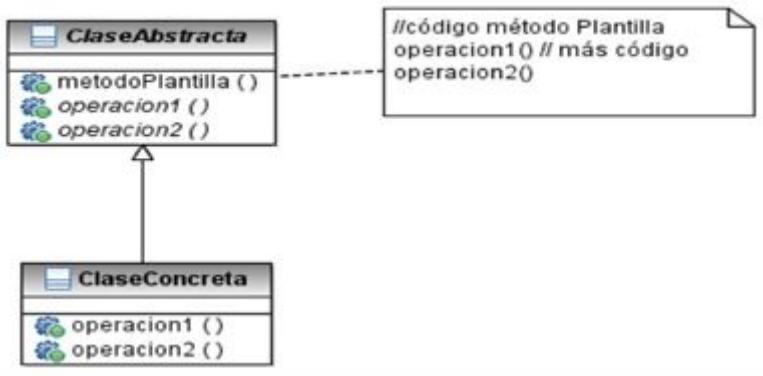
Define una familia de algoritmos relacionados, que se crean y configura desde el contexto, eliminando sentencias case y aumentando la cohesión de las clases involucradas. Parecido a State.

Template Method

Descripción: define el esqueleto de un algoritmo en una operación, delegando algunos pasos del mismo a sus subclases. Permite a las subclases redefinir ciertos pasos de un algoritmo sin cambiar la estructura del mismo.

Uso: fundamental para escribir código en un framework, permite implementar partes fijas de un algoritmo y dejar que las subclases implementen el comportamiento variable. Se controlan así los puntos de extensión de una clase.

Estructura:



Difiere de herencia común, en que el método plantilla es el que se usa generalmente, y no es redefinido, si no que lo que se redefine son las operaciones específicas de cada *ClaseConcreta*, que el método plantilla luego ejecuta.

Consecuencias:

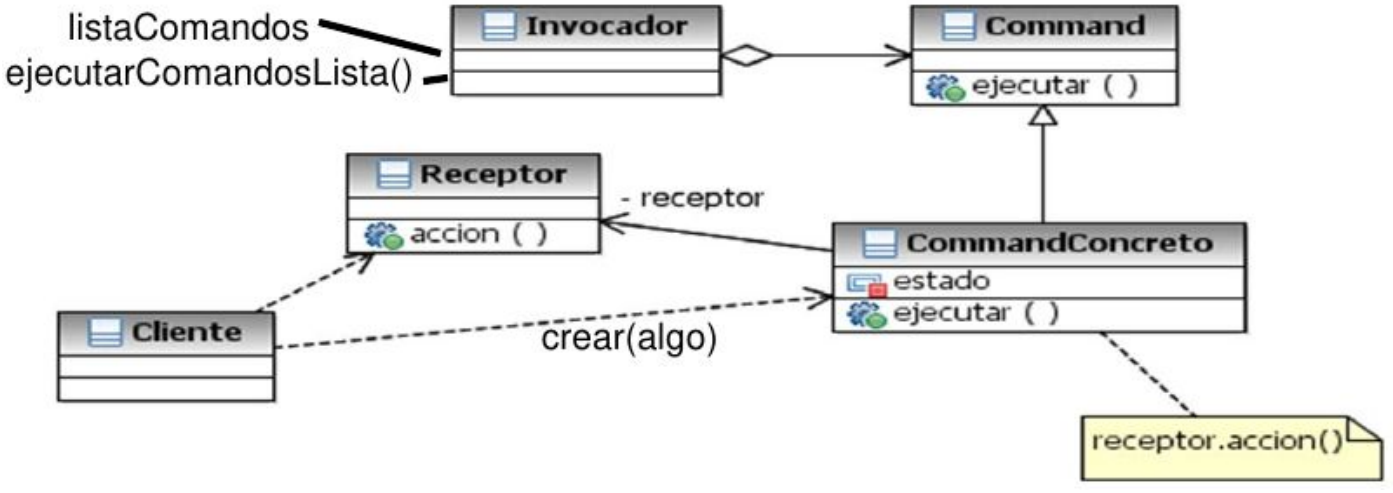
Mecanismo fundamental para reutilización, al permitir factorizar el comportamiento común en librerías de clases, al mismo tiempo que se controlan los puntos de extensión de una librería.

Command

Descripción: encapsula un mensaje como un objeto, permitiendo parametrizar los clientes con diferentes solicitudes, añadir a una cola las solicitudes o soportar funcionalidad undo/redo.

Uso: permite parametrizar objetos con una acción a realizar, especificar una función a ejecutar en un momento futuro, soportar undo/redo (al guardar el estado previo y posterior), recuperar fallos ante la ejecución fallida de la función, e implementar transacciones.

Estructura:



Consecuencias:

Desacopla al objeto que invoca la operación, del objeto que sabe cómo realizarla, al especificar un par receptor/acción. Es fácil añadir nuevos comandos.

Diseño de Persistencia (Bases de Datos - WF de Persistencia)

Para manejar la persistencia de un sistema (almacenar datos por fuera de lo que está en memoria de una aplicación) hacemos uso normalmente de una base de datos. Para hacer uso de una base de datos, utilizamos un sistema de gestión de bases de datos (DBMS), que maneje el almacenamiento en la base de datos, y nos abstraiga de los detalles de implementación del proceso de físico de almacenamiento.

El DBMS se debería encargar entonces, de proveer la **Concurrencia**, **Recuperación ante fallos**, y **facilidad de consulta** para con una DB.

Partiendo de los modelos de análisis y de requerimientos, podemos definir qué objetos van a ser persistentes, y luego de definir qué clases y variables de clase deben almacenarse en la DB, como normalmente el modelo de datos de las DB es relacional, debemos transformar una a una, las clases a tablas (con un OID, transformando sus relaciones a tablas, etc).
Esto se conoce como el **problema de impedancia**, y conlleva otras consideraciones:

- Qué hacer con los objetos complejos (que tienen funciones, por ejemplo)
- Qué hacer con la herencia?
- ¿Cómo relacionar los dos modelos sin generar un fuerte acoplamiento entre los mismos?
- Cómo modelar el bloqueo de datos para transacciones y la distribución de datos?.

Con respecto al mapeo de **herencia**, podemos mencionar 3 opciones:

- Definir una tabla para cada subclase, con todos los atributos de la tabla padre. Una solución muy rápida, pero que puede provocar inconsistencias.
- Definir una tabla padre, y tablas hijas con los atributos específicos, relacionadas a la tabla padre. Evita inconsistencias y redundancias, pero es más lenta, por necesitar más JOINs.
- Una tabla única, con todos los atributos posibles de las subclases, nullables. No recomendada.

Otro aspecto a considerar a la hora de definir las tablas de la DB es el nivel de normalización que deseamos tener. Normalizar nos garantiza cierta protección contra redundancias e inconsistencias, con la contracara de que usualmente las tablas normalizadas suelen requerir más JOINs, afectando esto a la performance. Esto último puede ser combatido con indexación, o desnormalizando algunas partes del modelo.

A la conversión de filas de tablas a objetos en memoria se lo llama **materialización** (que puede ser inmediata o perezosa), y a su inversa, **desmaterialización**.

ODBMS (DMBS orientado a Objetos)

Nos permiten almacenar objetos como tales, eliminando el problema de impedancia completamente. De esta manera, el diseño de la base de datos se integra durante el análisis y el diseño de la aplicación. Suelen incluso usar el lenguaje de programación, a diferencia de las DBMS relacionales.

Sin embargo, para ser considerados, deberían poder hacer una traducción 1 a 1 con el paradigma orientado a objetos, como tipos, herencia, objetos complejos, extensibilidad, y a esto se le debería sumar la identidad de objetos, para poder identificar cada objeto inequívocamente.

Diseño de Persistencia

Un esquema de persistencia es un conjunto reutilizable y extensible de clases e interfaces que prestan servicios a los objetos persistentes, que se encarga generalmente de la materialización y desmaterialización en un sistema, y de confirmar y deshacer transacciones.

El esquema de persistencia debería considerar:

Mapeo entre Clases y Tablas (1), la Identidad de los Objetos (2), el acceso al Servicio de Persistencia (3), las operaciones de Materialización y Desmaterialización (4), la caché (5), los estados de transacción de los objetos (6), cómo *volver atrás una transacción* (7), la lógica de materialización perezosa y la forma de acceder a los objetos usando proxies (8)

Super Objeto Persistente

Si decidimos tener un enfoque de correspondencia directa entre los modelos de la DB y los modelos del sistema, podemos crear un bloque abstracto que tenga la funcionalidad del esquema de persistencia, y que sea heredado por todos los bloques que necesiten almacenar información en la base de datos. Una alternativa a la herencia múltiple es hacer que el Súper Objeto Persistente sea una interfaz, que las clases de dominio tengan que implementar.
Para cualquiera de las dos formas, esto implica una disminución de la cohesión, y un gran acoplamiento.

Uso de patrones para el diseño del esquema de persistencia

1 Patrón Conversor (Mapper) o Intermediario (Broker) para Mapeo

Internamente, el subsistema podría tener una clase responsable de materializar y desmaterializar los objetos, con intermediarios (uno por cada clase a persistir) para hacer correspondencia directa entre registros y objetos.

2 Patrón Identificador de Objetos

Aparece ante la necesidad de contar con una forma consistente de relacionar objetos con registros, y asegurar que la materialización repetida no dé como resultado objetos duplicados. El patrón propone asignar un identificador de Objeto (OID) a cada registro y objeto (o proxy de objeto), para permitirnos identificar al objeto o registro en tabla inequívocamente.

3 Acceso al servicio de persistencia mediante Patron Fachada

Por medio de Fachada, podríamos decir que para la operación fundamental “obtener objeto dado un OID” se necesitaría entonces el OID del objeto, y el tipo de la clase del objeto a materializar. La fachada entonces delegaría la petición a los objetos de su subsistema.

4 Patrón Template Method y Cache

Teniendo en cuenta Cache, a la hora de materializar un objeto, la lógica suele ser algo como:
Si el objeto está en cache, retornar el objeto, caso contrario crear el objeto a partir del registro en DB, guardarlo en caché, y retornar el objeto.

Con Template Method, esta lógica se encapsula en una sola clase común el comportamiento de materialización.

5 Patron Gestion de Cache

Se usa para mejorar el rendimiento del subsistema, optimizando las operaciones de materialización y desmaterialización. Al materializar objetos, estos se guardan en cache con su OID y tipo como clave, por lo que el conversor primero busca en caché para evitar materializaciones innecesarias.

6 Patrón State y los Estados Transaccionales:

Los objetos persistentes pueden crearse, eliminarse o modificarse. Pero esto no implica que estas operaciones se realicen inmediatamente, si no que depende de que se ejecute una operación commit explícita, y debemos considerar que la operación a realizarse en la DB depende del estado del objeto en la aplicación.
Por ejemplo: Si el objeto está modificado en la aplicación “sucio” y ya existía en la DB “viejo” el objeto debería modificarse en la DB, y si por el contrario fuese “limpio” (sin modificar) y “nuevo” (no en DB) se debería crear en la DB.

7 Patrón Command y Transacciones

Una transacción es una unidad de trabajo (conjunto de tareas) cuya terminación es atómica (es decir, que se realizan todas las tareas o ninguna). Las tareas deben ordenarse antes de su ejecución, y se puede usar el patrón Command para wrapear cada tarea como un objeto con un método ejecutar() y un estado previo, para hacer una cola de operaciones a ejecutar, al mismo tiempo que se permite hacer rollback sobre los cambios.

8 Proxy Virtual y Materialización Perezosa

A veces es conveniente diferir la materialización de los objetos hasta que sea absolutamente necesario, por cuestiones de rendimiento. Para esto se suele utilizar un Proxy Virtual, que materialize al sujeto real cuando se le haga referencia por primera vez. Para esto se suele utilizar el OID del sujeto real, para identificar y recuperar el sujeto real.

WF de Implementación

Partiendo del Modelo de Diseño, consiste en implementar el sistema en términos de componentes, es decir, archivos de código fuente, scripts, archivos de código binario, ejecutable y similares, logrando desarrollar la arquitectura del sistema como un todo. Se inicia durante la fase de elaboración, durante la que se crea la base ejecutable, es el centro de la fase de construcción, y finalmente participa durante la fase de transición, para tratar defectos tardíos.

Propósitos

- Planificar las integraciones del sistema en cada iteración incremental.
- Distribuir el sistema asignando componentes ejecutables a nodos en el diagrama de despliegue.
- Implementar las clases y subclases encontradas durante el diseño
- Probar los componentes individualmente, para luego integrarlos, compilarlos y enlazarlos a uno o más ejecutables.

Artefactos

- **Modelo de Implementación:** describe cómo los elementos del modelo de diseño se implementan en términos de componentes, ficheros de código fuente, ejecutables, etc, y como se agrupan y relacionan utilizando el lenguaje/s de programación utilizados.
- **Componente**: equivalente a las clases de otros modelos. Algunos prototipos pueden ser *executable*, *file* (fichero con datos o código), *library*, *table* (de la db) y *document*.
- **Subsistema**: elemento de agrupación/organización de artefactos del modelo, con una interfaz, responsabilidades y dependencias definidas. Muy relacionado a los subsistemas de diseño del modelo de diseño (1 a 1). Se manifiesta de forma más concreta, como con paquetes de Java o proyectos, en Visual Basic.
- **Interfaz**: especifica las operaciones implementadas por componentes y subsistemas de implementación.
- **Descripción de la arquitectura:** vista del modelo de implementación, que muestra los componentes significativos arquitectónicamente, y como se descomponen el modelo de implementación en subsistemas.
- **Plan de Integración de construcciones:** El plan de integración de construcciones describe la secuencia de construcciones (una construcción es una versión ejecutable del sistema) necesarias en una iteración, definiendo la funcionalidad que se espera en cada construcción, y las partes del modelo de implementación afectadas.

Trabajadores

Arquitecto: responsable de la integridad del modelo de implementación, se asegura que el modelo sea correcto, completo y legible. Es responsable además de la arquitectura, y de la asignación de componentes a nodos.

Ingeniero de Componentes: responsable de definir y mantener el código fuente de uno o varios componentes y/o subsistemas.

Integrador de sistemas: encargado de planificar la secuencias de construcciones necesarias para cada iteración, y la integración cuando las construcciones han sido implementadas.

Actividades

El **Arquitecto** comienza **esbozando** los componentes claves del modelo de implementación. Luego el **Integrador planea** las integraciones necesarias de la presente iteración como una secuencia de construcciones (que es lo que hay que hacer), luego de esto, las clases y subsistemas son **implementados y probados** (pruebas unitarias) por el **Ingeniero de Componentes**, dando como resultado componentes, subsistemas e interfaces nuevas o refinadas de la construcción, para luego ser **integrado** por el **Integrador**, y se le hacen pruebas de integración, para luego continuar implementando la **siguiente** construcción e integrando hasta finalizar el plan de integración.

WF de Pruebas

En el flujo de trabajo de prueba verificamos el resultado de la implementación probando cada construcción, incluyendo tanto construcciones internas como intermedias, así como las versiones finales del sistema a ser entregadas a los clientes. Se centra durante la fase de elaboración (para probar la base arquitectónica) y construcción (para las pruebas del sistema en sí), en la fase de transición, se usa para encontrar defectos, y para realizar pruebas de regresión.

El propósito es planificar las pruebas necesarias en cada iteración, incluyendo las pruebas de integración y las de sistema. Las primeras son necesarias para cada construcción dentro de una iteración, las de sistema es la prueba final de cada iteración. Para esto se diseñan e implementan casos de prueba, que luego se ejecutan sobre las construcciones, obteniendo resultados a analizar.

Artefactos

- **Modelo de Pruebas:** describe principalmente como se prueban los componentes ejecutables en el modelo de implementación con pruebas de integración y de sistema. Es una colección de casos de prueba, procedimientos de prueba y componentes de prueba.
- **Caso de Prueba:** especifica una forma de probar el sistema, incluyendo el resultado esperado y las condiciones de prueba. Puede ser de caja negra o caja blanca, y se pueden probar muchas cosas, como la instalación, configuración (que funcione en diferentes configuraciones de sistema), negativas (intentando provocar una falla para revelar debilidades) o de estrés.
- **Procedimiento de pruebas:** es un documento que explica como realizar uno a varios casos de pruebas o partes de estos.
- **Componente de prueba:** automatiza uno o varios procedimientos de prueba o partes de ellos.
- **Plan de Prueba:** describe las estrategias, recursos y planificación de las pruebas para cada iteración y sus objetivos.
- **Defecto:** una anomalía del sistema.
- **Evaluación de Prueba:** es una evaluación de los resultados de los esfuerzos de prueba.

Actividades

El **Ingeniero de pruebas** determina el objetivo de la prueba a realizar, planificando el esfuerzo de prueba de cada iteración, describen entonces los casos de pruebas necesarios y los procedimientos de pruebas. Luego los **Ingenieros de Componentes**, crean componentes de pruebas para automatizar procedimientos de prueba, y luego, utilizando los casos, procedimientos y componentes de prueba, los **Ingenieros de Pruebas de Integración y de Sistema**, prueban cada construcción y detectan cualquier defecto que encuentren en ellos, sirviendo estos al **Ingeniero de Prueba** como información a evaluar y como retroalimentación para otros flujos de trabajo (como el de diseño e implementación).

- **(Ing. Pruebas) Planificar Prueba:** se describe una estrategia de prueba, los requerimientos para hacer la prueba (humanos y los sistemas necesarios) y planifica el esfuerzo de prueba.
- **(Ing. Pruebas) Diseñar Prueba:** se identifican y describen los casos de prueba (de integración, sistema y regresión), y los procedimientos de prueba necesarios para realizar los casos de prueba. Se intenta reutilizar procedimientos de prueba.
- **(Ing. Comp) Implementar pruebas:** se automatiza todo lo que se pueda los procedimientos de prueba.
- **(Ing. Pruebas Integración) Realizar Pruebas de integración:** para c/construcción de cada iteración.
- **(Ing. Pruebas Sistema) Realizar Pruebas de Sistema:** para cada iteración del sistema.
- **(Ing. Pruebas) Evaluar Pruebas:** se preparan métricas de nivel de calidad, y se comparan los resultados de la prueba con los objetivos establecidos.

Pruebas

Las pruebas pueden mostrar solo la presencia de errores, pero no su ausencia. **El propósito de la prueba es encontrar fallas.** El testing exitoso es el que encuentra defectos. Se puede **validar** *(se hace lo que se pide?)* o **verificar** *(¿lo que se hace está bien?)*.

Se puede probar durante el **desarrollo**, al finalizar una **versión**, o por medio de **usuarios** reales o potenciales.

WF de Despliegue

El propósito es volcar el producto finalizado a los usuarios. Se centra en la fase de transición. Esto incluye las siguientes actividades:

- Probar el software en el ambiente donde se va a operar.
- Empaquetar y distribuir el SW (puede incluir instalarlo, dependiendo la ocasión)
- Capacitar a los usuarios finales para su uso
- Migrar el SW existente (convertir bases de datos, migrar documentación, etc)

Artefactos

- El **artefacto principal es la *release***, que incluye el software ejecutable, todo lo que se necesite para instalarlo, notas de la release (changelogs, licencias), material de soporte (manuales) y de capacitación.
- Si es **manufacturado**, puede incluir el medio de almacenamiento, empaquetado y arte del producto, y la lista de materiales incluida en el mismo.
- Otros materiales que **se usan pero no se entregan** son los resultados de pruebas, de retroalimentación y un resumen de las pruebas de evaluación.

Actividades

1. **Crear el Plan de Despliegue**: define cómo y cuándo entregar el software, y cómo hacer para que el usuario final tenga toda la información necesaria para recibir el nuevo SW adecuadamente y comenzar a usarlo. Suele incluir una prueba beta del programa. Esta etapa requiere colaboración con el cliente, ya que muchas veces el problema no está en el software en sí, sino en el recibimiento del sistema de parte del cliente.
2. **Desarrollar el material de soporte y Capacitación** para instalar, usar, y mantener el sistema.
3. **Probar el producto en el lugar de desarrollo**: para ver cómo lo usa el cliente, nos da mejoras y correcciones de último momento.
4. **Armar la release:** dejar todo listo en términos de la release.
5. **Hacer un beta test de la release:** el cliente instala y lo usa durante un tiempo, proveyendo información sobre su performance y usabilidad, retroalimentando a los WF anteriores para próximas releases.
6. **Probar el producto en el lugar de uso:** para ver si el cliente puede instalarlo y usarlo sin contratiempos (acá ya se espera no encontrar ningún problema).
7. (opcional) **Empaquetar el producto**
8. (opcional) **Proveer acceso a la descarga del producto**

Evolución de Software

El desarrollo de SW no se detiene ante la entrega al cliente, sino que continúa a lo largo de la vida útil del mismo. Después de distribuir un sistema, inevitablemente este deberá modificarse, para mantenerlo útil, tanto por cambios empresariales como por las expectativas del usuario, que generan nuevos requerimientos.

Por esto, la ingeniería de SW se debe considerar como un proceso en espiral, con requerimientos, diseño, implementación y pruebas continuas, a lo largo de la vida del sistema. Esto comienza por crear la versión 1 del sistema, pero una vez entregada (e incluso antes en algunos casos) se comienza el desarrollo de la versión 2.

Al proceso de cambiar el software después de su entrega, cuando el que lo mantiene no es quien lo desarrolló (discontinuidad del espiral), se le llama Mantenimiento de Software, e incluye además de las actividades normales del desarrollo de SW, actividades de proceso adicionales (como la comprensión del SW).

Procesos de Evolución

Varían de acuerdo al tipo de SW mantenido, a la organización, y las habilidades de las personas que intervienen.

Pueden ser **informales**, donde las solicitudes de cambio provienen de conversaciones entre los usuarios del sistema y los desarrolladores, o **formales**, donde se realiza por medio de documentación estructurada.

Las **propuestas de cambio** pueden venir de requerimientos existentes aún no implementados, nuevos requerimientos, cambios de requerimientos, de reportes de bugs, y de nuevas ideas de mejora de SW de los propios desarrolladores.

El **proceso de evolucion,** incluye como actividades fundamentales el análisis de cambios, planificación de la versión, implementación del sistema y su liberación a los clientes.

En la etapa de **Implementación de Cambios**, el costo e impacto de los cambios se contraponen para saber si aceptarlos en la próxima iteración evolutiva o no. Sólo después se implementan, validan y se libera una nueva versión del sistema. En una primera iteración de esta etapa, puede involucrar una comprensión del programa, sobretodo si los desarrolladores del sistema original no son los responsables de implementar el cambio.

Caso especial son los **errores críticos**, o los cambios por legislación, donde no se analiza el costo beneficio del cambio, si no que simplemente se analiza el código, se implementa, valida y se entrega.

Leyes de Lehman

Aplicables a todos los tipos de sistemas de SW organizacional grandes, en donde los requerimientos se modifican para reflejar las necesidades cambiantes de la organización.

Algunas son (son 8 en realidad, pero se repiten o no tienen sentido):

- *Cambio continuo*: el sistema debe cambiar o se volverá progresivamente menos útil y satisfactorio
- *Declive Calidad:* la calidad del sistema disminuirá, a menos que se adapte a su entorno de funcionamiento.
- *Complejidad Creciente:* a medida que cambia, se volverá más complejo y necesitará más recursos para mantenerlo.
- *Estabilidad organizacional:* la velocidad de desarrollo es aproximadamente constante, e independiente de los recursos dedicados.
- *Conservación de la familiaridad:* debe mantenerse la familiaridad del sistema, para que ante cambios no se produzca una disminución de la capacidad productiva de la organización
- *Sistema de retroalimentación:* los procesos evolutivos deben ser retroalimentados para lograr mejores significativas del producto

Estrategias de Evaluación de Software

Mantenimiento

es el proceso de cambiar un sistema después de que este se entregó. Los cambios realizados al SW van desde los simples para corregir errores de codificación, hasta mejoras significativas o nuevas funcionalidades. Hay 3 tipos de mantenimiento:

- **Correctivo**: donde se corrigen errores de codificación, diseño o requerimientos.
- **Adaptativo**: donde se actualiza el sistema por algún cambio técnico del entorno, como el SO, el HW, etc.
- **Perfectivo**: donde se cambian o agregan nuevos requerimientos del sistema.

Mantener implicar además hacer **predicciones de mantenimiento**, para estimar las correcciones, cambios y mejoras y sus costos. Esto se hace teniendo en cuenta el **número y complejidad de interfaces**, el **número y complejidad de los requerimientos de sistema inestables** (propensos al cambio), y **los procesos de negocio** donde se use el sistema.

Reingeniería de Software

implica hacer ingeniería a partir de SW ya existente, con el propósito de mejorar su estructura y entendimiento. Implica crear un nuevo sistema a partir del antiguo sin cambiar su funcionalidad, por lo que entre las actividades a realizar se encuentran:

- Traducir el código fuente original/Hacer ingeniería inversa
- Mejorar la estructura del programa, y modularizar el código obtenido en el paso anterior.
- Hacer reingeniería de los datos del programa (para que se amolden al nuevo programa)
- Mejorar la arquitectura del programa.

Ventajas vs Sustitución:

- No hay errores de especificación de sistema o de requerimientos.
- Suele ser más barato que hacerlo de 0

Desventajas vs Sustitución:

- Si el sistema es funcional, hacerlo OO o viceversa no es posible.
- Grandes cambios no pueden hacerse de forma automatizada.
- El mantenimiento de sistemas producto de reingeniería es mas dificil (estructura fea)

Refactorización

Es el proceso de hacer mejoras a un programa para frenar la degradación mediante el cambio, **sin agregar funcionalidad**. Esto implica modificar un programa para mejorar su estructura, reducir su complejidad, o hacerlo más fácil de entender.

A diferencia de la reingeniería donde se hace un nuevo sistema a partir de otro más antiguo, la refactorización es un proceso continuo de mejoramiento del sistema original, en donde la intención es mejorar la estructura y el código, que aumentan los costos de mantenimiento.

Cosas a refactorizar: Sentencias case (switch), Datos mal organizados, código duplicado, métodos largos.

Administración de Sistemas Heredados

Implica evaluar los sistemas heredados para elegir la estrategia de mantenimiento a seguir, teniendo en cuenta el **valor empresarial**, el **valor técnico** del sistema (su estructura y entorno) y la **calidad técnica** del mismo (su documentación, la facilidad de mantenimiento, y que tan confiable es).

- Si la calidad y el valor de negocio es bajo, se desecha.
- Si la calidad es baja y el valor de negocio es alto, se hace reingeniería.
- Si la calidad es alta, y el valor es bajo, y el cambio es caro, se desecha.
- Si la calidad es alta y el valor es alto, se mantiene.