

Dos objetos son **ortogonales** si y solo si un cambio en un componente no produce afectaciones a otro componente de sistema

Ley de Conway

Mel Conway publicó "How Do Committees Invent?", en el cual exploraba la relación entre la estructura organizacional de una empresa y el resultado del diseño de un sistema. Una frase en ese mismo escrito se convirtió en lo que hoy conocemos como la **Ley de Conway**. Esta dice:

Organizations which design systems... are constrained to produce designs which are copies of the communication structures of these organizations.

Esta establece que hay una relación directa entre la comunicación que ocurre dentro de una organización y el diseño de la arquitectura de sistemas.

El diseñar una pieza de software o cambiar la arquitectura de las aplicaciones en la empresa implica hacer cambios en las estructuras de comunicación entre personas o equipos. Si las organizaciones buscan hacer cambios en el software sin cambiar los flujos de comunicación, no se obtendrán resultados.

*Se pueden diferenciar tres tipos de flujos de comunicación que ocurren dentro de una empresa. El primer flujo es el organizacional, conocido como organigrama; el segundo es el informal y el tercer flujo es la estructura de creación de valor, este último es donde el trabajo se hace.

*La comunicación no asegura realizar un trabajo de mejor calidad, hay que elegir cuándo y cómo va a ocurrir.

¿Qué es Diseño de Sistemas?

El **diseño de sistemas** es pensar, modelar y diagramar una solución conceptual que resuelva un problema; es el plano general de un sistema. Incluye la evaluación de las distintas soluciones alternativas y la especificación de cada una de ellas. No existe una solución a un único problema, sino que debe existir un abanico de posibilidades, de las cuales se seleccionara la solución más adecuada al problema, teniendo en cuenta las restricciones del contexto.

*En **diseño de sistemas** se debe tener en cuenta en los niveles de hardware, esto involucra pensar a niveles de diseño de arquitectura.

*Se debe pensar en cómo estará conformado el sistema, es decir los subsistemas, cómo se comunican, sus responsabilidades, etc.

Definición formal:

"Es la estrategia de alto nivel para resolver problemas y construir una solución. Ésta incluye decisiones acerca de la organización del sistema en subsistemas, la asignación de subsistemas a componentes de hardware y software, y decisiones fundamentales, conceptuales y de política, que son las que constituyen un marco de trabajo para el diseño detallado."

Diseño de Sistemas =/ Diseño de Software

En **diseño de software** pensamos soluciones a niveles de código mientras que en **diseño de sistemas** pensamos soluciones a niveles macro.

Análisis y Diseño

*Los ingenieros piensan soluciones

*Las soluciones deben ser las adecuadas, entendiendo las restricciones del contexto

*Se debe comprender el problema o necesidad y el contexto para crear un buen producto

Requisitos y Requerimientos

→ Son todas las necesidades y deseos pedidos por el cliente y las personas involucradas en el Sistema.

→ Todas las funcionalidades, características y restricciones que debería tener el sistema.

-> **Requerimientos:** es la necesidad/deseo de un cliente

-> **Requisitos:** es una circunstancia o condición necesaria para algo.

Estos deben ser:

- ✓ No ambiguos
- ✓ Concisos
- ✓ Consistentes
- ✓ Completos
- ✓ Alcanzables

Los **REQUISITOS/REQUERIMIENTOS FUNCIONALES** son aquellos que responden a la necesidad del usuario/actores.

Los **REQUISITOS/REQUERIMIENTOS NO FUNCIONALES** son aquellos que corresponden a las propiedades del sistema (rendimiento, seguridad, etc.)

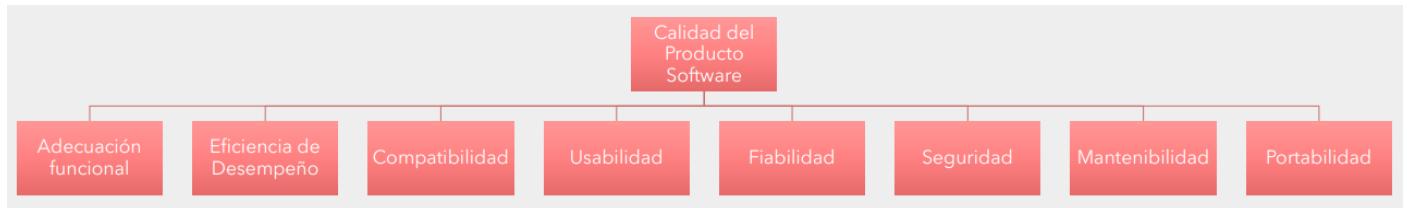
Clase 05/04

La calidad la **PUEDO MEDIR**, no la puedo **PROBAR**

Calidad de Software

Para la norma ISO 25000 se define como **CALIDAD DE SOFTWARE** como el grado en que un producto satisface los requisitos de sus usuarios. Son estos requisitos los que se encuentran representados en el modelo de calidad de ISO 25010.

El modelo de calidad del producto definido por la ISO 25010 se encuentra compuesto por ocho características que, a su vez, estas características se subdividen en varias subcaracterísticas.



ADECUACIÓN FUNCIONAL (FUNCIONALIDAD)

La **ADECUACIÓN FUNCIONAL** es la capacidad del producto software para proveer las funciones que satisfacen las necesidad explícitas e implícitas cuando el software se utiliza bajo condiciones específicas.

→ **COMPLETITUD FUNCIONAL**: grado en el cual el conjunto de funcionalidad cubre todas las tareas y objetivos esperados por el usuario.

→ **CORRECIÓN FUNCIONAL**: capacidad del producto o sistema para proveer resultados correctos con un nivel de precisión requerido.

→ **PERTINENCIA FUNCIONAL**: capacidad del producto software para proporcionar un conjunto apropiado de funciones para tareas y objetivos de usuario especificados.

EFICIENCIA DE DESEMPEÑO (PERFORMANCE)

La **EFICIENCIA DE DESEMPEÑO** representa el desempeño relativo a la cantidad de recursos utilizados bajo condiciones determinadas.

→ **COMPORTAMIENTO TEMPORAL**: los tiempos de respuestas y procesamiento y los ratios de throughput de un sistema cuando se lleva a cabo sus funciones bajo condiciones determinadas.

→ **UTILIZACIÓN DE RECURSOS**: las cantidades y tipos de recursos utilizados cuando el software lleva a cabo su función bajo condiciones determinadas.

->Ejemplo: memoria, GPU.

→ **CAPACIDAD**: grado en que los límites máximos de un parámetro de un producto o sistema software cumplen con los requisitos, esto quiere decir que nuestro sistema mantiene un grado de eficiencia bajo cierta capacidad, establece un tope máximo.

COMPATIBILIDAD

La **COMPATIBILIDAD** es la capacidad de 2 o más sistemas o componentes de relacionarse e intercambiar información bajo el mismo entorno hardware o software.

→ **COEXISTENCIA**: capacidad del producto software para coexistir con otro software independiente, compartiendo recursos comunes

→ **INTEROPERABILIDAD**: capacidad de 2 o más sistemas o componentes para utilizar información e intercambiarla.

Los sistemas deben ser compatibles e internamente deben tener entidades interoperables; las entidades interoperables deben ser tratadas por diferentes sistemas de la misma manera (ejem.: una dirección) y esto provoca que, a su vez, esos sistemas sean interoperables. Para ello, se establece un protocolo estándar o comunicación común.

2 o más sistemas son interoperables si se entienden y la forma de comunicación es estándar a nivel modelo, es decir a un nivel abstracto.

USABILIDAD

La **USABILIDAD** es la capacidad del producto software de ser entendido, aprendido, usado y atractivo al usuario. Se basa en la interfaz gráfica.

→ **ADECUACIÓN**: capacidad del producto de dar a entender al usuario si el software es adecuado para sus necesidades.

→ **CAPACIDAD DE APRENDIZAJE**: capacidad del producto de enseñar al usuario a aprender su uso.

→ **CAPACIDAD PARA SER USADO**: capacidad del producto que permite al usuario operar y controlarlo con facilidad.

→ **PROTECCIÓN CONTRA ERRORES DE USUARIO**: capacidad del sistema para proteger a los usuarios de hacer errores.

→ **ESTÉTICA DE LA INTERFAZ**: capacidad de la interfaz de usuario de agrandar y satisfacer la interacción del usuario.

→ **ACCESIBILIDAD**: capacidad del producto que permite que sea utilizado por usuarios con determinadas características y discapacidades.

FIABILIDAD

La **FIABILIDAD** hace referencia a la capacidad de un sistema o componente para desempeñar funciones específicas, esto quiere decir que es la capacidad del software para realizar funciones de manera consistente y sin fallas.

→ **MADUREZ**: capacidad del producto software para evitar fallas como resultado de errores en el software.

Actualmente, se intenta manejar los errores y controlar las fallas, en vez de evitarlas.

→ **DISPONIBILIDAD**: capacidad de un sistema o componente para estar operativo y accesible para su uso cuando se lo requiere, es decir que es la capacidad del sistema o componente de estar “andando” al momento que se lo necesite.

→ **TOLERANCIA A FALLOS**: capacidad de un producto software de mantener un nivel de funcionamiento en caso de errores de software o incumplimiento. Mantener el sistema funcionando a pesar de que ocurra algún fallo.

→ **RECUPERABILIDAD**: capacidad del producto software para reestablecer un nivel de funcionamiento y recuperar los datos afectados en caso de falla.

SEGURIDAD

La **SEGURIDAD** se define como capacidad de un sistema de proteger información y datos para que personas o sistemas no autorizados no puedan leerlos o modificarlos.

-> CIA: confidencialidad, integridad y disponibilidad (confidentiality, integrity and availability)

→ **CONFIDENCIALIDAD**: capacidad de protección contra el acceso de datos e información no autorizados.

→ **INTEGRIDAD**: capacidad del sistema o componente para prevenir accesos o modificaciones no autorizados a datos o programas de ordenador.

→ **NO REPUDIO**: capacidad del software de demostrar que las acciones o eventos realizados por un usuario sean verificables y no puedan ser negados.

→ **RESPONSABILIDAD**: capacidad de rastrear de forma única las acciones de una entidad.

→ **AUTENTICIDAD**: capacidad de demostrar la identidad de un usuario o recurso.

Evolutivo -> Extender
Perfectivo -> Mejorar
Correctivo -> Corregir algo

MANTENIBILIDAD

La **MANTENIBILIDAD** es la capacidad que presenta un producto software para ser modificado de forma efectiva y eficiente con el fin de satisfacer las necesidades evolutivas, correctivas o perfectivas.

→ **MODULARIDAD**: capacidad de un software de permitir que un cambio en un componente tenga un impacto mínimo en los demás.

→ **REUSABILIDAD**: capacidad de un producto de ser utilizado en más sistemas o en la construcción de otros productos.

→ **ANALIZABILIDAD**: es la facilidad con la que podemos medir el impacto de un cambio sobre el resto del software, diagnosticar las deficiencias, las causas de fallos o identificar partes a modificar.

→**CAPACIDAD PARA SER MODIFICADO**: es la capacidad de un producto que permite que sea modificado de forma efectiva y eficiente sin causar problemas.

→**CAPACIDAD PARA SER PROBADO (“TESTEABILIDAD”)**: es la facilidad con la que podemos establecer criterios de prueba para un sistema o componente y la facilidad con la que podemos llevar a cabo esas pruebas cumpliendo con los criterios establecidos.

PORTABILIDAD

La **PORTABILIDAD** se define como la capacidad de un producto o componente de ser transferido de un entorno a otro sin problemas.

→**ADAPTABILIDAD**: es la capacidad del producto software para ser adaptado a diferentes entornos.

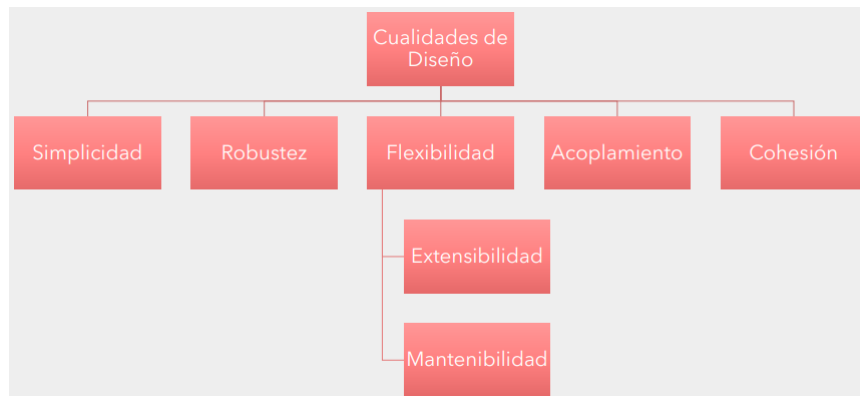
→**CAPACIDAD PARA SER INSTALADO**: es la facilidad con la que podemos instalar y/o desinstalar el producto de forma exitosa en nuestro entorno.

→**CAPACIDAD PARA SER REEMPLAZADO**: es la capacidad del producto para ser utilizado en lugar de otro producto en el mismo entorno y con el mismo propósito.

Cualidades de Diseño

Las **CUALIDADES DE DISEÑO** son una re-clasificación de criterios que nos sirven para analizar distintas propuestas y tomar decisiones de diseño más acertadas.

No son los únicos criterios: la experiencia y el conocimiento de quien esté realizando el proceso de diseño serán elementos determinantes.



ACOPLAMIENTO

El **ACOPLAMIENTO** se define como el grado de dependencia entre dos componentes, es decir la cantidad de conocimiento que un componente tiene sobre el otro.

Mientras más conozca o dependa un componente a otro, será mayor el acoplamiento de ese componente al otro. Cuanto mayor sea el acoplamiento entre dos componentes, los cambios o errores de uno de ellos impactarán en mayor medida sobre el otro componente.

->**Minimizar** el acoplamiento, no extinguirlo.

->Mantener un acoplamiento bajo, mejora la mantenibilidad de un sistema.

->Se mejora la modularidad (subcaracterística de la mantenibilidad), ya que esta consiste en permitir cambios en un componente sin impactar demasiado a los demás.

Si minimizamos el acoplamiento se puede:

->Mejorar la mantenibilidad

->Aumentar la reutilización, ya que, al ser un sistema con componentes desacoplados, podemos reutilizar esos componentes para otros sistemas

->Evitar que un defecto de un componente se propague a otro

->Evitar tener que inspeccionar o modificar muchos componentes cuando se realiza la modificación en un solo componente

COHESIÓN

→Un **COMPONENTE COHESIVO** suele tener todos sus elementos abocados a resolver el mismo problema.

→La **COHESIÓN** se define como la cantidad de responsabilidades que le damos un componente.

->Mientras más responsabilidades tenga un componente, menor será su cohesión.

IDAÑEZ, LUCIA MARÍA 🐱

->Encontrar buenas abstracciones para mejorar la asignación de responsabilidades y, con ello, la cohesión.

ALTA COHESIÓN, BAJO ACOPLAMIENTO

SIMPLICIDAD

KISS → Keep it simple, stupid / Keep it Short and Simple

->Evitar cualquier complejidad innecesaria.

->No complejizar de manera innecesaria algo que podría hacerse de manera más sencilla.

YAGNI → You aren't gonna need it

->No agregar funcionalidades nuevas que no apunten a resolver la problemática actual.

->Diseñar pensando en la extensibilidad, pero sin hacer algo que nadie nos pidió.

ROBUSTEZ

La **ROBUSTEZ** establece que ante el mal uso/uso inadecuado del sistema por parte del usuario u otros sistemas, o ante fallas internas:

→El sistema no debe fallar o comportarse de forma rara.

→El sistema debe reportar los errores y volver a un estado consistente, esto quiere decir que el sistema debe avisar al usuario qué está pasando y volver a como estaba antes de la falla/error.

→El sistema debe facilitar la detección de la causa del problema; debe ser fácilmente analizable (subcaracterística de mantenibilidad).

->Para hacer que nuestro sistema sea analizable se utiliza el mecanismo de **logs**.

FLEXIBILIDAD

La **FLEXIBILIDAD** es la capacidad de reflejar cambios en el dominio de manera simple y sencilla.

→**EXTENSIBILIDAD**: capacidad de agregar nuevas cosas con poco impacto.

->EXTENSIBILIDAD - -> Mantenimiento EVOLUTIVO, agregado de características a mi sistema para mejorarlo

→**MANTENIBILIDAD**: capacidad de modificar las características existentes con el menor esfuerzo posible.

->Mantenimiento CORRECTIVAS o PERFECTIVAS

Patrones de Diseño (O.O.)

“Son descripciones de clases y objetos relacionados que están adaptados para resolver un problema de diseño general en un contexto determinado”

-Erich Gamma, Richard Helm, John Vlissides y Ralph Johnson

Definición de la cátedra:

Los **PATRONES DE DISEÑO** son soluciones conocidas a problemas conocidos y reiterados en el mundo de desarrollo de software. El objetivo de los patrones es reutilizar la experiencia de quienes trabajaron con problemas similares y encontraron una buena solución.

ESTRUCTURA DE LOS PATRONES DE DISEÑO:

- ✓ Propósito
- ✓ Motivación
- ✓ Participantes
- ✓ Colaboraciones
- ✓ Consecuencias
- ✓ Implementación
- ✓ Usos conocidos
- ✓ Patrones relacionados

PARTES ESENCIALES DEL PATRÓN DE DISEÑO:

→**NOMBRE**: comunica el objetivo del patrón de diseño en una o dos palabras.

→**PROBLEMA**: describe el problema que el patrón soluciona y su contexto; indica cuándo se aplica el patrón, sin embargo, no podemos garantizar que, si se dan las condiciones para aplicar el patrón, lo usemos.

→**SOLUCIÓN**: indica cómo resolver el problema en términos de elementos, relaciones, responsabilidades y colaboraciones. La solución debe ser abstracta para poder ser aplicada en muchas situaciones.

IDAÑEZ, LUCIA MARÍA 🐱

→**CONSECUENCIAS:** indica los efectos de aplicar la solución.

CLASIFICACIÓN DE LOS PATRONES DE DISEÑO:

→**CREACIONALES:** se utilizan para resolver problemas al momento de instanciación o creación de un objeto complejo, o no, tomando decisiones dinámicas en momento de ejecución. Atacan los problemas de instanciación de objetos o configuración de objetos en momentos de ejecución.

→**COMPORTAMIENTO:** se utilizan para resolver cuestiones, complejas o no, de interacción entre objetos en momento de ejecución. Los patrones de comportamiento siempre van a tratar de interacción de objetos.

→**ESTRUCTURALES:** se resuelven cuestiones complejas de generación o utilización estructuras complejas que no están dentro dominio.



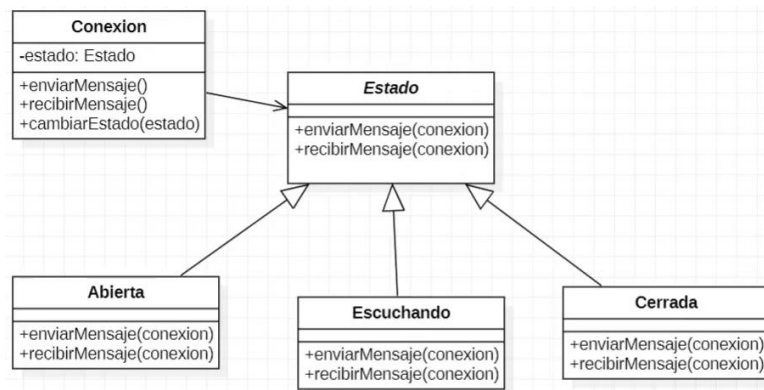
Video Uri

Patrón State

PATRON STATE es un patrón de COMPORTAMIENTO.

Motivación: Se requiere modelar la conexión de un dispositivo a la red de Internet teniendo en cuenta los siguientes estados: CONECTADO, ESCUCHANDO Y CERRADO.

DIAGRAMA DE CLASES:



¿Qué hace?

->Genera una abstracción (una clase) por cada posible estado que pueda tener un objeto.

->Define transiciones entre los posibles estados.

Se sugiere su utilización cuando:

->El comportamiento de un objeto depende de su estado y este estado puede variar en tiempo de ejecución.

->Un método está lleno de condicionales (sentencias if u otra) que dependen del estado del objeto. Estos estados suelen estar representados por varios atributos de distintos tipos: primitivos o enumerados.

COMPONENTES:

→Interface (o clase abstracta) State: Define las firmas de los métodos que dependen del estado del objeto principal, en este caso la clase abstracta se llama **ESTADO** donde se encuentran las firmas **enviarMensaje(conexion)** y **recibirMensaje(conexion)**.

IDAÑEZ, LUCIA MARÍA 🐱

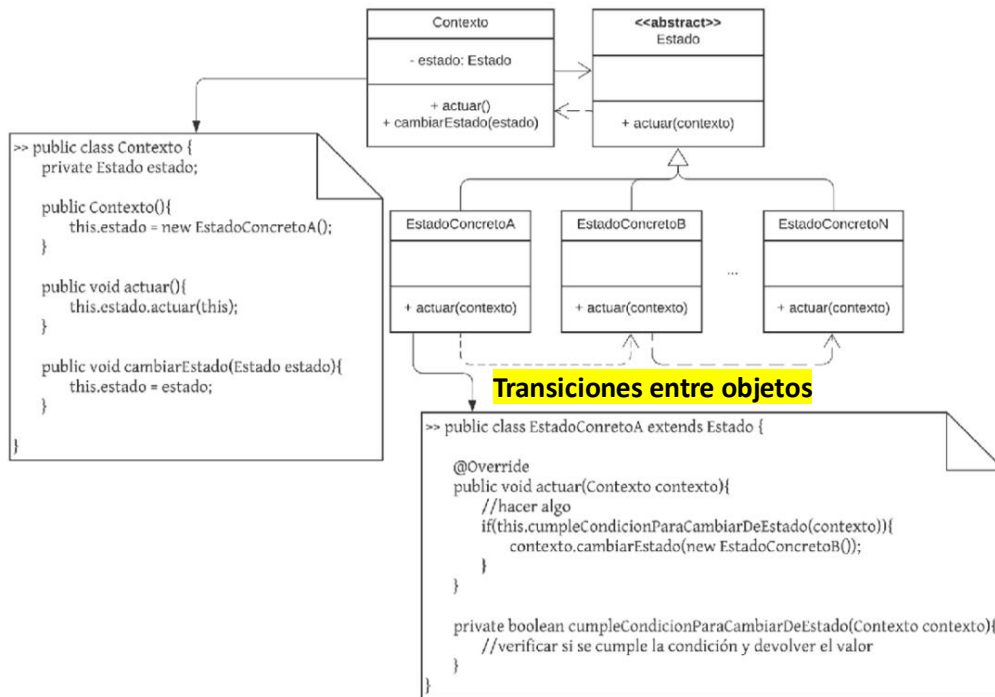
→ Clases de estados concretas: clases que implementan la interface State (o que heredan de ella, si ésta fuera clase abstracta), es decir, que tienen la implementación real de los métodos, en nuestro caso nuestras clases concretas con **ABIERTA**, **ESCUCHANDO** y **CERRADA**. Son los estados posibles del objeto p

→ Contexto: clase que tiene referencia a la interface/clase abstracta State,

Ventajas del uso del Patrón State:

- ✓ Mayor cohesión a la clase Contexto
- ✓ Mejora la mantenibilidad debido a que el comportamiento por cada estado es fácilmente localizable.
- ✓ Extensibilidad para incorporar nuevos estados con nuevo comportamiento

Ventaja: si llega a “fallar” algún estado va a ser más fácil encontrar el problema, por ende, mejora la analizabilidad



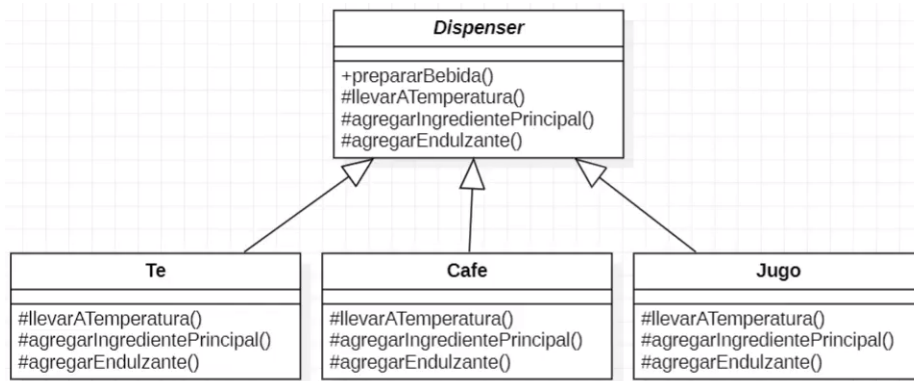
Patrón Template Method

PATRON TEMPLATE METHOD es un patrón de COMPORTAMIENTO.

Motivación: Se desea crear un dispenser para bebidas. Para preparar las mismas se debe llevar el agua a la temperatura adecuada y luego dependiendo de la bebida se precede de forma distinta:

- >Té: se desea calentar el agua a 100, luego servir el té y finalmente agregar azúcar
- >Café: una vez calentada el agua a 100, colocar el café y servir con edulcorante
- >Jugo: se debe enfriar el agua, luego agregar el polvo para que diluya en agua y servir con Stevia

DIAGRAMA DE CLASES:



¿Qué hace?

->Define el esqueleto de un algoritmo, estableciendo los pasos que sí o sí se deben implementar.

En nuestro caso ese **esqueleto de algoritmo** es **prepararBebida()** el cual debe implementar los pasos **llevarATemperatura()**, **agregarIngredientePrincipal()** y **agregarEndulzante()**.

Se sugiere utilizar este patrón cuando:

IDAÑEZ, LUCIA MARÍA 🐱

->Varias abstracciones tienen los mismos pasos y el mismo orden, pero cada una de estas abstracciones tiene una implementación diferente.

->Cuando 2 o más objetos son utilizados de forma polimórfica, esto quiere decir que esos objetos ejecutan el mismo algoritmo, respetando los pasos, pero cada uno realiza implementaciones diferentes.

COMPONENTES:

→ Clase abstracta: define el método plantilla, es decir que define el esqueleto del algoritmo, qué se va a ejecutar, en qué orden, con qué parámetros, etc.

→ Clases concretas: define los algoritmos que se van a ejecutar cuando se ejecuten los pasos.

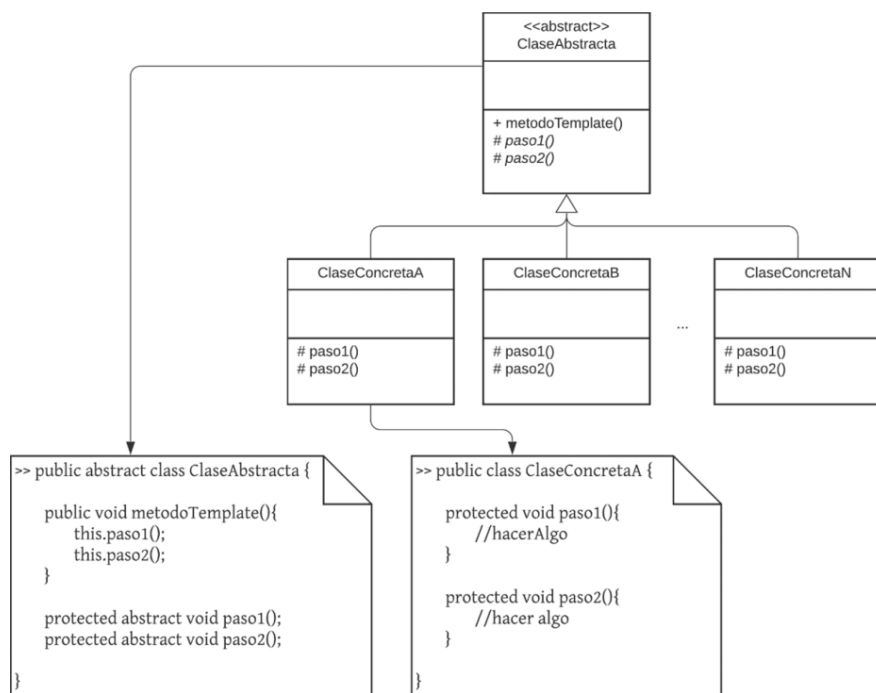
Consideraciones:

->El método template puede no ser void, es decir que puede devolver algo, y los pasos también pueden no ser void.

->Pueden existir sentencias intermedias entre los “pasos” que tengan comportamiento en común para todas las clases concretas, es decir que se pueden usar métodos (concretos) que fueron definidos en la clase abstracta.

Ventajas del Patrón Template Method:

- ✓ Mejora la mantenibilidad, ya que el comportamiento de los “pasos” es fácil de localizar.
- ✓ Tiene una alta cohesión en todas las clases involucradas.
- ✓ Permite una extensibilidad para agregar nuevas clases concretas que definan nuevo comportamiento para los pasos.



Video Ezequiel

MOCKEO DE OBJETOS

Para poder MOCKEAR los objetos vamos a utilizar la biblioteca MOCKITO que nos va a ayudar a definir algunos objetos “impostores”, estos objetos nos van a permitir generar algunas respuestas en nuestros test.

Una vez hecha la dependencia de MOCKITO estamos en condiciones de comenzar a mockear nuestros objetos, esto quiere decir que podemos comenzar a generar nuestros objetos impostores con el fin de realizar test sin que estén implementadas las cosas.

->Primero hacemos un `import static org.mockito.Mockito.*;`

->Al hacer esto estamos importando todos los MÉTODOS DE CLASE de la clase Mockito

Con el método **mock()** lo que vamos a hacer es generar un objeto Mock u objeto Impostor que va a tener que respetar una interfaz, para esto le debemos pasar la clase o interfaz a respetar:

```
CalculadorDeDistancia calculadorDeDistancia = mock(CalculadorDeDistancia.class);
```

Al llamar al método `mock()` y decirle que quiero que me genere una instancia que respete esta interfaz, ya vamos a obtener una instancia que cumple con esa interfaz y va a estar guardada, en nuestro caso, en **calculadorDeDistancia**, ésta va a ser mi instancia impostora.

Ahora le damos el comportamiento con el método **when()** y le vamos a decir que cuando alguien llame a ese objeto y, en particular, llame a un método, esto haga una determinada cosa.

```
CalculadorDeDistancia calculadorDeDistancia = mock(CalculadorDeDistancia.class);
when(calculadorDeDistancia.distanciaEnMetrosEntre(medrano, campus)).thenReturn(10100F);
```

En este caso hicimos que cuando (**when(...)**) alguien llame a

calculadorDeDistancia.distanciaEnMetrosEntre(medrano, campus) se debe retornar, **thenReturn(10100F)**, 10100.

```
CalculadorDeDemora calculadorDeDemora = mock(CalculadorDeDemora.class);
when(calculadorDeDemora.demoraAproximadaEnMinsParaRecorrer(unosMetros: 10100F)).thenReturn(30.0);
```

En este otro caso, hicimos un objeto mockeado (una instancia) de **CalculadorDeDemora**, este queda guardado en **calculadorDeDemora** y, como es una instancia “mentirosa” de la interfaz **CalculadorDeDemora**, le damos el comportamiento con el método **when()** que, en este caso, lo que va a hacer es que cuando alguien llame a **calculadorDeDemora.demoraAproximadaEnMinsParaRecorrer(10100F)** debe retornar, **thenReturn(30.0)**, 30.

Clase 19/04 + Video Uri

Patrón Strategy

PATRÓN STRATEGY es un patrón de COMPORTAMIENTO.

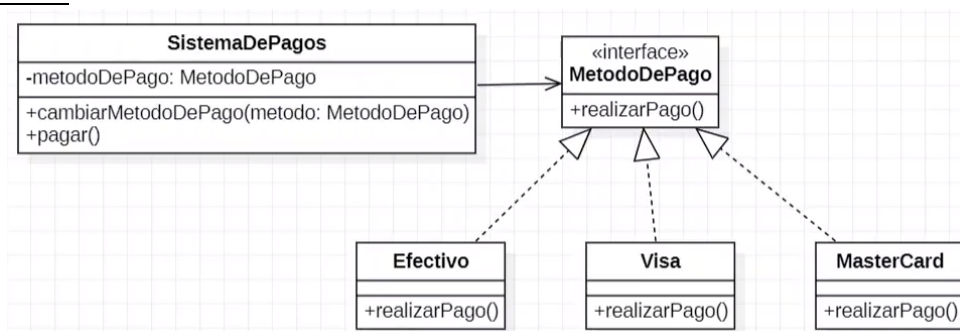
Motivación: Se requiere modelar un sistema de pagos de un e-commerce de forma tal que los clientes puedan realizar los mismos a través de:

- >Efectivo
- >Visa
- >MasterCard

Además, se debe tener en cuenta que en un futuro se pueden agregar nuevos medios de pago.

→ Algo para tener en cuenta es que debemos SIEMPRE prever que nuestros dominios deben ser extensibles.

DIAGRAMA DE CLASES:



En este diagrama, tenemos el **SistemaDePagos** que tiene un atributo que es el método de pago a utilizar **metodoDePago: MetodoDePago** y dos métodos **cambiarMetodoDePago(método: MetodoDePago)** y **pagar()**. El método **cambiarMetodoDePago(método: MetodoDePago)** lo único que va a hacer es cambiar el valor del atributo que se tiene por aquel que se pasa por parámetro al método. El método **pagar()** lo que va a hacer es llamar al método **realizarPago()** de cada uno de los métodos de pago.

A su vez, tenemos la interfaz **MetodoDePago** que tiene la firma del método **realizarPago()**, este método lo van a implementar cada una de las clases que tengan esa interfaz, como **Efectivo**, **Visa** y **MasterCard**; cada uno de estos métodos de pago implementan el algoritmo de cómo se va a realizar ese pago, estas clases tienen el cuerpo del método mientras que la interfaz solo tiene la firma.

¿Qué hace?

IDAÑEZ, LUCIA MARÍA 🐱

->Encapsula las distintas formas (o algoritmos) de resolver el mismo problema en diferentes clases, esto quiere decir que, si tenemos distintas formas de resolver un mismo problema, encapsulamos esas “distintas formas” en clases (por cada forma de resolver el problema, se crea una clase).

->Permite intercambiar en momento de ejecución la forma en que un tercero resuelve un problema.

Se sugiere su utilización cuando:

->Se requiere que un objeto realice una misma acción, pero con un algoritmo distinto o de una forma distinta.

->Existen muchas formas de realizar una misma acción, pero con distintos pasos en el mismo objeto.

->Hay que configurar en momento de ejecución la forma en que un objeto realiza una acción.

COMPONENTES:

→Interface o clase abstracta Strategy: define la firma del método que se utilizará como estrategia de resolución de algún problema.

→Clases de estrategias concretas: clases que implementan (o que heredan de ella, si fuera una clase abstracta) Strategy, es decir son aquellas clases que definen el algoritmo, o el cuerpo, de aquel método que se utiliza como estrategia de resolución de problema que se encuentra en la clase o interfaz Strategy.

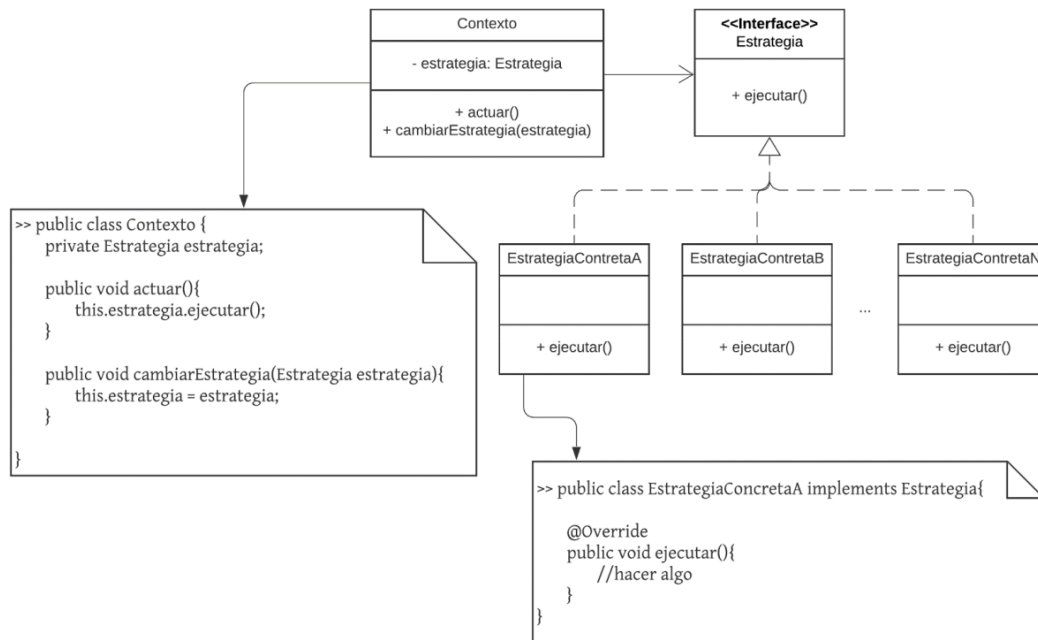
→Contexto: clase que tiene referencia a la interface/clase abstracta Strategy.

Ventajas del Patrón Strategy:

- ✓ Mayor cohesión a la clase Contexto.
- ✓ Mejora la mantenibilidad, ya que el comportamiento de cada algoritmo es fácilmente localizable.
- ✓ Permite una extensibilidad para incorporar nuevos algoritmos/formas de realizar las acciones.

PATRÓN STATE VS. PATRÓN STRATEGY

Mientras que en el patrón **STATE** había **transiciones** entre los distintos estados, en el patrón **STRATEGY** **no hay ningún tipo de transición**. A su vez, en el PATRÓN STRATEGY los algoritmos de cada clase concreta no se conocen entre ellos, es decir que todos aquellos algoritmos definidos por las clases que implementan la interfaz (o heredan) no se deben conocer entre ellos.

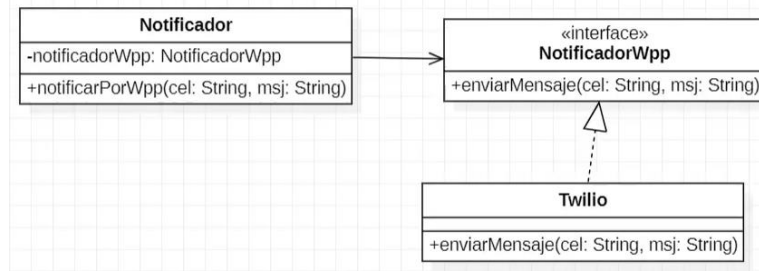


Patrón Adapter

PATRÓN ADAPTER es un patrón ESTRUCTURAL.

Motivación: Se requiere hacer uso de una librería para enviar mensajes de WhatsApp a todos los usuarios del sistema.

DIAGRAMA DE CLASES:



En este sistema, tenemos un Notificador que posee un atributo **notificadorWpp: NotificadorWpp** y un método **notificarPorWpp(cele: String, msj: String)** que notifica por WhatsApp a un celular con un determinado mensaje. También tenemos una interfaz **NotificadorWpp** que define un método **enviarMensaje(cele: String, msj: String)** y una librería que, en este caso, es Twilio, sin embargo el día de mañana podría dejar de ser Twilio esta librería, pero ¿qué pasa? Nuestro sistema va a estar adaptado a Twilio y esto es lo que NO DEBE PASAR, no debemos adaptar nuestro sistema para que utilice una ÚNICA librería, debemos buscar un ADAPTADOR entre mi sistema y esas librerías, a esto se lo denomina EL PATRÓN ADAPTER.

Volviendo a nuestro sistema, nuestro método **notificarPorWpp(cele: String, msj: String)** lo que hace es llamar al método **enviarMensaje(cele: String, msj: String)** que va a estar en el valor del atributo **notificadorWpp**, es decir el mensaje que va a estar en **NotificadorWpp**. Entonces, en caso de que el valor de **notificadorWpp** sea Twilio, el mensaje que se va a ejecutar va a ser el **enviarMensaje(cele: String, msj: String)** que está en la librería de Twilio. De esta manera, si queremos, luego, agregar otra librería solo debemos hacer que esa librería implemente la interfaz de **NotificadorWpp** y con eso podremos cambiar el valor del atributo **notificadorWpp** de **Notificador** por la nueva librería.

¿Qué hace?

->Encapsula el uso (llamadas/envío de mensajes) de la clase que se quiere adaptar en otra clase que concuerda con la interfaz requerida.

Se sugiere su utilización cuando:

->Se tiene que seguir adelante con el diseño y/o implementación (código) sin conocer exactamente cómo, quién y cuándo se resolverá una parte necesaria, y solo se conoce la responsabilidad que tendrá esa parte.

->Esto quiere decir que sabemos qué se quiere resolver, pero no cómo, entonces ponemos un ADAPTER.

->Se tiene que usar una clase que ya existe, pero su interfaz no concuerda/no matchea con la interfaz que se necesita.

COMPONENTES:

→**Interface Adapter:** define la firma del método que se utilizará como “puente” de acoplamiento a nuestro dominio.

→**Clases de Adapters concretos:** clases que implementan la interface Adapter. Guardan referencia o hacen uso de las clases Adaptadas. Llamamos a los métodos que van a resolver nuestro problema.

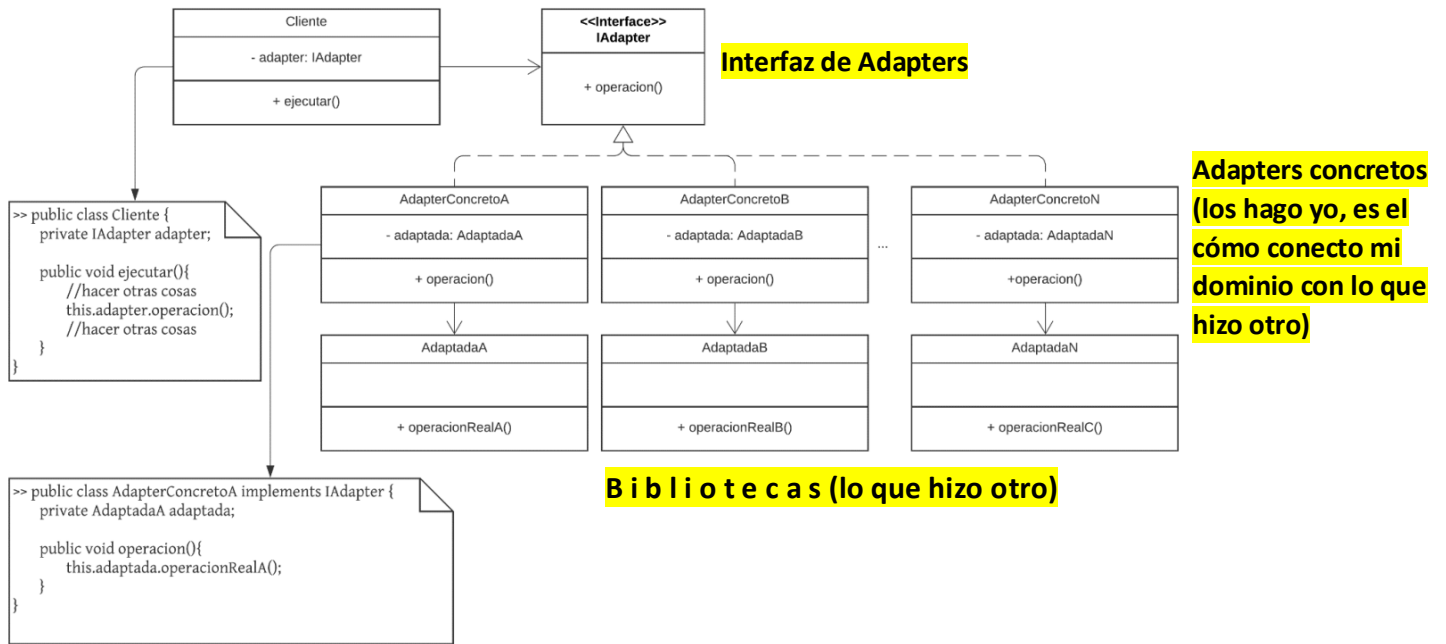
→**Clases adaptadas:** clases externas al dominio, o no, poseen firmas incompatibles o que todavía no conocemos.

Estas clases no se pueden/deben tocar.

→**Cliente:** clase que tiene la referencia a la interface Adapter, sus objetos van a hacer uso de las funcionalidades que les brinda esta clase Adapter.

Ventajas del Patrón Adapter:

- ✓ Mejor mantenibilidad ya que no tiene acoplamiento entre el cliente y la clase adaptada.
- ✓ Mejor cohesión de la clase cliente debido a que delega cierto comportamiento en el adapter.
- ✓ Mayor facilidad de testeo ya que se podría implementar un mock de AdapterConcreto.



Componentes Stateless y Stateful

“El estado de una aplicación hace referencia a su condición o cualidad de existir en un momento determinado. Es su capacidad de estar. Que un sistema tenga estado depende del tiempo durante el cual se registra interacción con él y de la forma en que se debe almacenar esa información.”

-Red Hat – Sistemas con y sin estado

Componentes Stateless

Un proceso, aplicación o un componente sin estado se refiere a los casos en que éstos están aislados. Estos componentes no almacenan información sobre las operaciones anteriores ni se hace referencia a ellas, por ende, cada operación que se ejecute se realiza desde 0, como si fuera la primera vez.

OBJETOS

Los objetos Stateless no deberían tener estado interno; o si lo tienen, el estado no debería influir en el funcionamiento cuando se realicen las distintas peticiones que podría recibir el objeto en cuestión.

Los objetos Stateless son reutilizables, ya que no dependen de otros objetos para sobrevivir, sino que tienen razón de existencia.

Componentes Stateful

Las aplicaciones, procesos o componentes con estado son aquellos a los que podemos volver una y otra vez y van a recordar “quiénes somos” y “qué hicimos”.

Se realizan con el contexto de las operaciones anteriores y la operación actual puede verse afectada por lo que ocurrió antes.

OBJETOS

Los objetos Stateful generalmente no son reutilizables, ya que, por estar un poco acoplados a un objeto, no podrían servir para que otro objeto realice operaciones con ellos.

Clase 03/05 – Video Uri

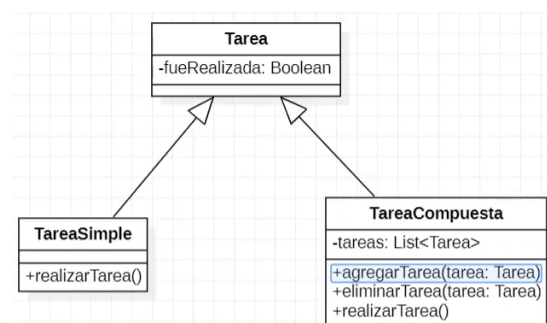
Patrón Composite

PATRÓN COMPOSITE es un patrón ESTRUCTURAL.

Motivación: Nos solicitaron el desarrollo de una aplicación que le permita a los usuarios organizar su lista de tareas diarias, para eso se decidió separar a las tareas en:

- Tareas simples: que involucran una sola acción
- Tareas compuestas: involucran dos o más tareas que puede ser simple o, a su vez, compuestas

DIAGRAMA DE CLASE:



En este caso, la clase **Tarea** está modelada como una clase **abstracta** y de ella heredan **TareaSimple** y **TareaCompuesta**. La **TareaSimple** implementa el método **realizarTarea()** que va a cambiar el valor del atributo **fueRealizada**; la clase **TareaCompuesta** posee un atributo **tareas: List<Tarea>** esto va a generar una estructura de ÁRBOL donde la **TareaCompuesta** posee varias **Tarea** hijas y estas tareas hijas pueden ser **Simple**s o **Compuestas**. La clase **TareaCompuesta** le patea la pelota (delega la acción) de **realizarTarea()** en su colección de Tareas.

¿Qué hace?

- >Compone objetos en estructura de árbol
- >Permite que un cliente trate de forma polimórfica a un objeto en particular y a un conjunto combinado de ellos (es decir, tanto el objeto como el conjunto de objetos entienden el mismo mensaje)
- >Dos objetos o más van a hacer uso de un mismo método, pero de forma distinta.

Se sugiere su utilización cuando:

- >Se requiere que un tercero utilice de forma indistinta (polimórfica) a un objeto particular a un conjunto compuesto de ellos.
- >Se debe permitir que un objeto esté compuesto por varios objetos simples o por varios objetos compuestos.

COMPONENTES:

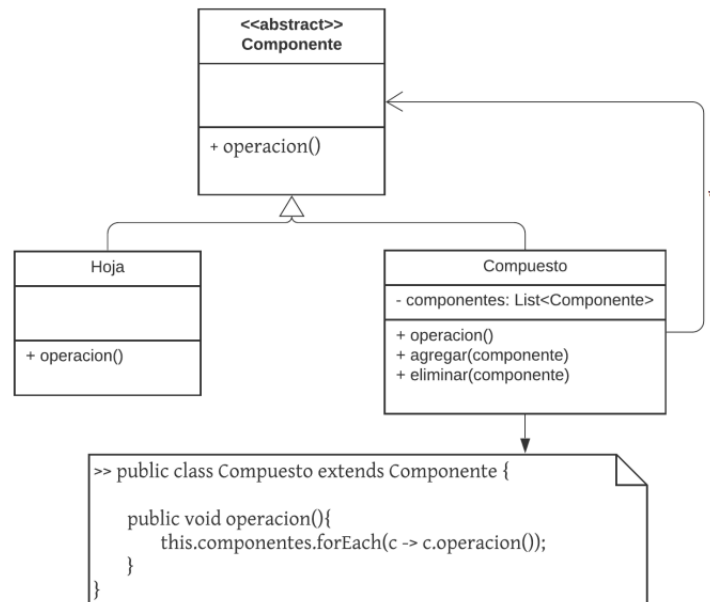
→ Interface/Clase Abstracta Componente: define las firmas de los métodos que son comunes entre los elementos simples y los elementos compuestos. Un tercero debe conocer la interfaz para que utilice de forma polimórfica a los elementos simples y compuestos.

→ Hoja: objetos simples que cumplen con la interfaz del Componente. Contienen todo el algoritmo/acción de los métodos comunes de la clase/interfaz componente.

→ Compuesto: clase que cumple con la interfaz Componente. Contiene una colección de Componentes, esa colección puede contener objetos simples (hojas) u objetos compuestos. Al recibir un mensaje, delega el trabajo en los Componentes de la colección -esto quiere decir que no implementa la acción, sino que pateja esa pelota a los Componentes de la colección-, luego junta las respuestas, las procesa y otorga una respuesta final.

Ventajas del Patrón Composite:

- ✓ Mejora la extensibilidad para crear nuevos compuestos y hojas que agreguen comportamiento.
- ✓ Simplifica la forma de tratar a una estructura de objetos compuestos.



Patrón Decorator

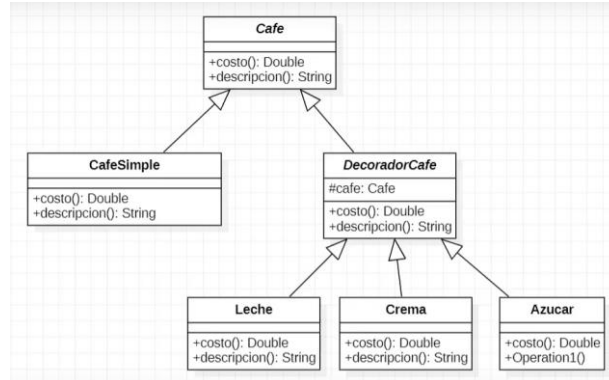
PATRÓN DECORATOR es un patrón ESTRUCTURAL.

Motivación: Se requiere desarrollar una máquina expendedora de café, donde cada usuario puede elegir el tipo.

Teniendo en cuenta que por el momento hay tres tipos:

- >Con leche: el precio es el original más un 3%
- >Con crema: el precio es el original más un 5%
- >Con azúcar: el precio es el original más un 1%

DIAGRAMA DE CLASES:



Como podemos ver tenemos la clase abstracta **Cafe** que tiene el método **costo()** y **descripcion()** que son los métodos que van a implementar las clases hijas **CafeSimple** y **DecoradorCafe** (también es una clase abstracta).

DecoradorCafe tiene un atributo **cafe: Cafe**, el cual va a ser **CafeSimple**, y los métodos (**costo()** y **descripcion()**) en su implementación lo que van a hacer es llamar a esos métodos **costo()** y **descripcion()** del atributo **Cafe**.

De la clase **DecoradorCafe** heredan nuestros DECORADORES CONCRETOS, en este caso **Leche**, **Crema** y **Azúcar**, los cuales también implementan los métodos **costo()** y **descripcion()** pero **agregan algo más** a cada método; por ejemplo, en el caso de **Leche** el método **descripcion()**, al estar el atributo **cafe** referenciado a **CafeSimple**, normalmente devolvería el string "Cafe" pero el método de la clase **Leche** le va a agregar "Café con leche", ya que esa clase va a interceptar ese mensaje **descripcion()** y agregarle algo, agrega una DECORACIÓN, lo mismo sucede con las otras clases y el resto de los métodos.

¿Qué hace?

->Agrega funcionalidades a un objeto sin romper su interfaz

->Intercepta mensajes que "vienen" del padre y agrega algo más al comportamiento de ese mensaje. Existen diferentes formas de interceptar un mensaje:

->Intercepción parcial: mantiene el comportamiento del mensaje que viene del padre y agrega algo más

->Intercepción total: no hace uso del comportamiento del padre, lo reemplaza totalmente

Se sugiere su utilización cuando:

->Se requiere agregar y/o quitar funcionalidades/responsabilidades a un objeto en momento de ejecución

->Existen condicionales que restringen/amplían las acciones que realiza un objeto frente a la recepción de un mensaje

Cuando tenemos un objeto y queremos cambiar un poco el comportamiento, pero no queremos romper su interfaz, el patrón Decorator es una buena opción.

COMPONENTE:

→Clase Abstracta/Interface Componente: contiene la firma del método a interceptar.

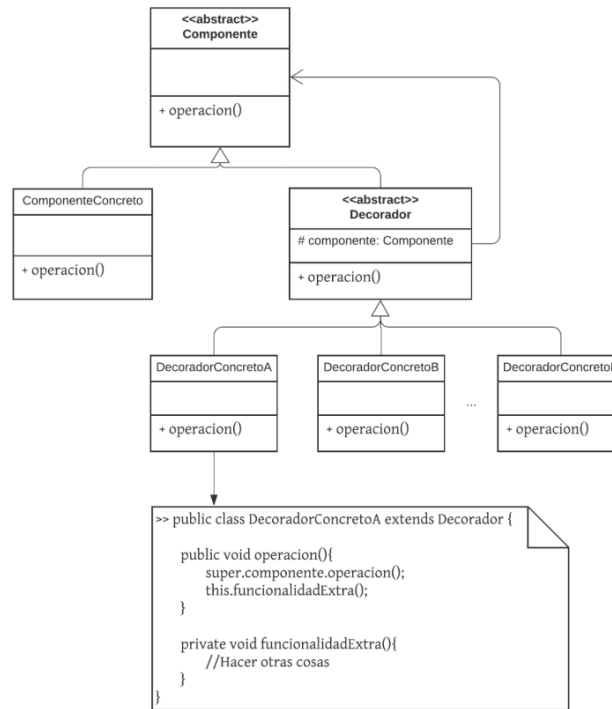
→Clase Componente Concreto: clase que hereda (o implementa interface) de la clase Componente y define el comportamiento del método que se quiere interceptar. También conocida como "clase a decorar".

→Clase Abstracta Decorator: clase abstracta que tiene la referencia a la clase Componente. Envuelve a la clase a decorar. Define la/s firma/s del método a interceptar por los decoradores concretos.

→Decoradores concretos: clases que heredan de la clase Decorator, se encargan de interceptar los métodos/mensajes, agregando o quitando funcionalidades.

Ventajas del patrón Decorator:

- ✓ Mejor cohesión en el componente concreto y en los decoradores concretos
- ✓ Permite agregar nuevas responsabilidades sobre componentes ya creados (extender responsabilidades)



Clase 10/05

Principios SOLID

Principios básicos de Programación Orientada a Objetos y Diseño de Sistemas que nos ayudan a obtener mejores diseños mediante una serie de reglas o principios.

Nos ayudan a evitar la generación de “código sucio”

Los principios SOLID son:

- >Single Responsibility Principle
- >Open Closed Principle
- >Liskov Substitution Principle
- >Interface Segregation Principle
- >Dependency Inversion Principle

SOLID -> Single Responsibility Principle

->Principio de responsabilidad única

Una clase debe tener una única responsabilidad. **Clases Cohesivas.**

Esta responsabilidad debe estar encapsulada por la clase y todos sus servicios deben estar alineados con esa responsabilidad.

Se debe evitar la clase “Dios” propiciando la alta Cohesión.

SOLID -> Open Closed Principle

->Principio de Abierto Cerrado

Las entidades deben estar abiertas para la expansión (mayor extensibilidad), pero cerradas para su modificación.

->No se debe modificar un código al momento de extenderlo, se debería extenderlo sin tocar el código

Se basa en la implementación de herencias y el uso de interfaces para resolver el problema.

Se sugiere evitar la utilización excesiva de “switchs” y propiciar el polimorfismo entre objetos.

SOLID -> Liskov Substitution Principle

->Principio de sustitución de Liskov

Si S es un subtipo de T, entonces los objetos de tipo T pueden ser reemplazados por los objetos tipo S sin alterar ninguna de las propiedades del sistema.

->Se deberían poder intercambiar la clase padre y clase hijo sin problemas (no debería romper nada)

->Siempre que tengamos una herencia debemos tipar todas las instancias de la clase hija como lo más general que tengamos, es decir con la interfaz de la superclase más general que tengamos.

->Las clases hijas no deben tener más métodos que los que estén definidos en su clase padre.

Se debería utilizar correctamente la herencia y las realizaciones de Interfaces.

SOLID -> Interface Segregation Principle

->Principio de segregación de interface

Los clientes de un componente sólo deberían conocer del componente aquellos métodos que realmente usan y no aquellos que no necesitan usar.

Este principio separa las interfaces que son muy grandes en interfaces más pequeñas y específicas, de manera que los clientes sólo tengan que conocer los métodos que les interesan.

Muchas interfaces cliente específicas son mejores que una interfaz de propósito general.

->Al separar las interfaces un componente implementa solo aquellas interfaces que usa y el cliente conoce solo lo mínimo necesario

->Un objeto debería conocer de otro lo mínimo que necesita, el mínimo mensaje que necesita

Se debería propiciar un diseño orientado a interfaces para mantener el acoplamiento entre clases al mínimo posible y evitar generar interfaces con muchos métodos (extensas)

SOLID -> Dependency Inversion Principle

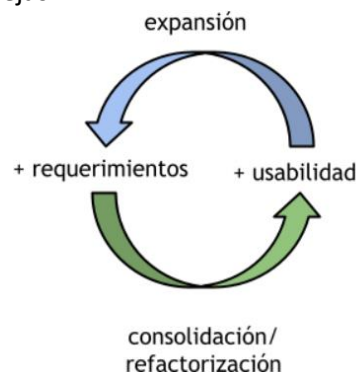
Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos módulos deben depender de sus abstracciones.

Es una forma de desacoplar los componentes de alto nivel de los componentes de bajo nivel, de manera que sea posible la reutilización de los componentes de alto nivel al usar componentes de bajo nivel distintos.

Se sugiere utilizar inyectores de dependencias

Refactoring

REFACTORIZAR no es incorporar funcionalidades nuevas ni optimizar nuestro código. **REFACTORIZAR** es cambiar la estructura del código para hacerlo más simple y extensible; esto involucra muchas acciones como cambiar el nombre de una clase o método, o cosas más complejas.



¿QUÉ BUSCA EL REFACTOR?

->Métodos cortos y con nombre bien definidos; intention revealing, que revelen exactamente el propósito para el que fueron creados

->Métodos y clases con responsabilidades claras y bien definidas (mayor cohesión)

->Respecto a las variables de instancia: hay que evaluar el costo-beneficio de tener en **variables valores que pueden calcularse**

→Esto es importante ya que, si vamos a estar calculando varias veces un mismo valor, es conveniente tenerlo almacenado en una variable con el fin de ahorrar memoria/recursos y mejorar la performance

->No tener god objects ni managers, estos son objetos que roban responsabilidades que les corresponden a otros objetos

->Es preferible tener objetos chicos antes que un objeto grande con muchas responsabilidades, sobre todo si los objetos chicos pueden intercambiarse, esto tiene que ver con generar objetos polimórficos que puedan ser intercambiables para un tercero

->Evitar relaciones bidireccionales innecesarias

REFACTORIZAR ≠ REDISEÑO ≠ REINGENIERÍA

*Los IDE ayudan bastante con el tema de refactorizar

IDAÑEZ, LUCIA MARÍA 🐱

Code Smells

CODE SMELLS -> “olores o aromas de código”

Los **smells** no indica que aquello que diseñé o implementé esté mal, sino que son señales de que algo se puede mejorar. Nosotros debemos ser capaces de identificar aquellos “olores” en nuestro código para poder mejorar nuestro diseño.

-> Suele suceder que al intentar mejorar un code smell, generemos otro code smell

Algunos de los Code Smell son:

CÓDIGO DUPLICADO

-> **Falta de herencia o delegación**

No queremos duplicar la misma idea en el código. Hay dos lemas que nos acompañan en esto:

→ Once and only once: hacer las cosas una sola vez

→ Don't repeat yourself (DRY)

El **código duplicado** también aplica para el diseño en general. Dos formas de evitar este smell en diseño es la composición y la herencia.

MÉTODOS LARGOS

->Descomposición inadecuada

Un método largo podemos descomponerlo en varias partes, ya que este método puede tener muchas responsabilidades, es poco cohesivo.

Al método largo se lo debe descomponer en varias partes, de cada una de esas partes se debe:

- >identificar el objetivo que cumple esa parte del método;
- >implementar esa abstracción: ponerle un nombre representativo y encontrar qué objeto/método dentro del mismo objeto es responsable de ese objetivo

Los métodos generados pueden ser utilizados en otro contexto.

Delegar el método original en otros métodos que nos permitan entender qué es lo que hace.

GOD CLASS

->Demasiadas responsabilidades

Una god class es una clase que hace de todo, tiene muchas responsabilidades y conoce información de muchos objetos, esto provoca que se vea impactada ante cualquier cambio.

- >No hacer nunca una clase "Sistema" o parecida

Para solucionar este smell se puede:

→**Generar nuevas clases**: si el comportamiento de clase es grande y cumple con varias funciones al mismo tiempo, deberíamos romper a esa God Class y separarla en otras clases que cumplan con un solo propósito (clases altamente cohesivas). -> **Alta cohesión** →Principio de Responsabilidad única

→**Generar subclases**: si parte del comportamiento de la God Class se puede implementar de diferentes maneras, podríamos convertir a nuestra God Class en una clase abstracta y generar subclases/clases hijas que definan el comportamiento específico necesario. →Principio de sustitución de Liskov

→**Generar interfaces**: si lo que queremos es brindar una lista de operaciones con diferentes comportamientos que el cliente puede utilizar, poder generar una/varios interfaces y crear clases concretas que cumplan con dicha interfaz e implementen las diferentes funcionalidades. →Principio de segregación de interfaces

Si la God Class es responsable de una interfaz gráfica, se debe comenzar a pensar en el patrón MVC, e ir descomponiendo las responsabilidades entre las diferentes capas involucradas.

PARÁMETROS LARGOS

->Objeto perdido

Puede darse cuando varios algoritmos se combinan en un solo método o cuando nos estamos olvidando de alguna abstracción del dominio.

Si un objeto va a brindar un servicio a otro, lo mejor sería que la cantidad de parámetros del método no cambie.

Por ejemplo:

```
>> CatalogoDePelis
```

```
public void getPelicula(String nombrePeli, String autor, int anio, int versión, int  
estrellas){...}
```

Este método está muy sujeto a esos parámetros y cualquier cambio en alguno de ellos afectaría a todo nuestro código. Si queremos agregar un parámetro más, estamos rompiendo con el cliente que utiliza este servicio.

Una posible solución es:

```
public void getPelicula(BusquedaPelicula peliBuscada){...}
```

CADENA DE MENSAJES

Si un objeto envía mensajes de esta forma:

```
a.b().c().d();
```

Va contra el consejo de **Tell, don't ask** y contra la **Ley de Demeter (don't talk to strangers, only talk to friends)**, donde un objeto solo debería enviar mensajes:

- >a sí mismo
- >a objetos que conoce (atributos)
- >a objetos que recibe como parámetro
- >a objetos que instancia

IDAÑEZ, LUCIA MARÍA 🐱

**Esto no se cumple a rajatabla*

HERENCIA RECHAZADA

La **herencia rechazada** se puede dar cuando una clase sobrescribe todo un método base definido por su padre sin importar lo que este método le haya dicho que tenía que hacer o cuando una clase hereda métodos que en realidad no debería tener.

La herencia es rígida:

->obliga a definir más métodos de los que son realmente necesarios

->en la mayoría de los lenguajes la herencia es simple, lo cual limita todo a un solo punto de vista

Posible solución:

→Utilizar composición por sobre herencia

LAZY CLASS

->**Clase que no hace nada o hace muy poco**

Suele darse en el caso de la extrema delegación o porque el componente fue diseñado “por las dudas”.

Para evitar caer en este smell:

→Pensar en YAGNI (“No lo necesitas”) para no agregar funcionalidad hasta que sea necesario

→Eliminar de nuestro diseño las clases PLD (“por las dudas”)

NOMBRES DE VARIABLES CORTOS

El nombre de las variables debe representar su funcionalidad, debemos ser EXPRESIVOS.

NOMBRES DE MÉTODOS LARGOS

Si nos cuesta elegir/formular un nombre representativo para un método, y queda muy largo, es una señal de que hay mucha lógica comprendida dentro del método.

CÓDIGO MUERTO

El **código muerto** es el código “por las dudas” que nos marea a la hora de ver la responsabilidad que tiene asignada una clase.

Solución:

→Borrar todo el código muerto y diseñar de modo extensible pero cerrado al cambio

PRIMITIVE OBSESSION

El **primitive obsession** suele darse al querer representar con ints, booleans, Strings o Enums todas aquellas cosas que podrían ser objetos con comportamiento. Los Enums nos llevan a tener condicionales en lugar de poder trabajar con objetos polimórficos.

Ejemplo:

```
Public enum Estacion { VERANO, OTONIO, INVIERNO, PRIMAVERA }
Public void actuar(){
    swith(estacion) {
        Case(VERANO) ..... break;
        Case(OTONIO)... break;
    }}
}}
```

Solución:

→Buscar abstracciones que nos permitan trabajar polimórficamente

DATA CLASSES

->**Solo accesorios**

Una **data class** es una clase que sólo representa una estructura de datos, es fácil reconocerla porque solo tiene getters y setters.

Estas clases son una mala práctica de diseño, ya que no es bueno pensar en separa un objeto en: atributos y comportamiento, solo por el hecho de separa la estructura de datos y los procesos que acceden a esa estructura.

Existen excepciones como los VALUE OBJECTS (como un Mail o una coordenada) y los objetos que modelan los parámetros que enviamos a un objeto (como BusquedaPelicula en el ejemplo de Long Parameter List).

MÉTODOS FUERA DE LUGAR E INTERMEDIARIO

>> MiembroDeComunidad

```
public void unirseAComunidad(Comunidad comunidad){
    comunidad.agregarMiembro(this);
}
```

Si no accedemos a ningún atributo ni tampoco utilizamos comportamiento interno, posiblemente estamos ubicando mal el método: ¿por qué no enviar directamente el mensaje a “Comunidad”?

NOTA: **En objetos no se modela la realidad sino una abstracción de esta**

1. ¿De quién es la responsabilidad de...?
2. ¿Quién tiene la información mínima e indispensable para...?

Entities vs. Value Objects

Las entidades y los value objects son abstracciones para el dominio.

La diferencia entre entidades y value objects consiste en que, por un lado, las **ENTIDADES** son las **ABSTRACCIONES PRINCIPALES Y/O RELEVANTES PARA EL DOMINIO**, poseen un sentido de existencia propio; mientras que los **VALUE OBJECTS** representan **ABSTRACCIONES ACCESORIAS** que solamente **GUARDAN DATOS**, estos datos **SOLAMENTE TIENEN SENTIDO DE EXISTENCIA SI SE LOS CONTEXTUALIZA CON ALGUNA OTRA ENTIDAD**.

Por ejemplo:

Una clase **ALUMNO** tiene un atributo **TELEFONO**, el cual podría estar representado por una instancia de clase “**TELEFONO**”, esta clase puede tener atributos como número, código de área y el tipo (móvil, fijo, etc).

La clase **Teléfono** instanciada sola, no tiene sentido de existencia propio si no está atada a un **Alumno**, ya que la clase **Teléfono** es **ACCESORIA**, nosotros tomamos una decisión que, para no tener esos atributos (número, código área y tipo) sueltos en el **Alumno**, los llevamos a una clase **Teléfono** que únicamente va a almacenar esos datos.

TIPOS DE IGUALDAD

→ IGUALDAD DE REFERENCIA (IDENTIDAD)

Dos objetos son iguales si son exactamente el mismo, es decir si están referenciando a la misma porción de memoria.

→ IGUALDAD ESTRUCTURAL (EQUIVALENCIA)

Dos objetos son iguales si todos los valores que tienen sus atributos, en cierto momento, coinciden.

No son iguales porque tienen residen en espacios de memoria diferentes, pero en ese momento son iguales por equivalencia.

→ IGUALDAD DE IDENTIFICADORES

La entidad va a tener algún atributo que lo identifique unívocamente, un identificador natural o ficticio.

Dos objetos son iguales si tienen exactamente el mismo valor para su identificador.

Las entidades tienen identificadores, los value object no tienen identificador porque son accesorias.

VIDA ÚTIL

→ ENTIDADES

Las entidades viven **A LARGO PLAZO**.

Pueden tener un historial de lo que les sucedió y cómo cambiaron durante su vida.

Tienen sentido de existencia propio.

→ VALUE OBJECT

Son **DESECHABLES**, tienen una vida útil de cero o “muy corta”.

No tienen sentido de existencia propio, sino que están atados a las entidades.

Se crean y destruyen con facilidad, es decir que son intercambiables.

INMUTABILIDAD

→ ENTIDADES

Las entidades son **mutables**, es decir que sus atributos pueden cambiar a lo largo del tiempo.

→ VALUE OBJECTS

Los value object son **inmutables**, es decir que sus atributos no pueden cambiar de valor.

Si se necesita cambiar el valor de algún atributo, se desecha la instancia y se crea un value object nuevo.

Se les agrega **final** a los atributos de los value object.

Si no se puede hacer que un value object sea inmutable, entonces no es un value object

¿ENTIDAD O VALUE OBJECT?

Nosotros tomamos la decisión de modelar alguna abstracción como una clase por motivos como:

- > Pueden existir muchos casos para esa abstracción y no sabemos cuántos son
- > La frecuencia al cambio puede ser “no baja”, es decir que puede ser a cada rato se requieran agregar nuevos casos
- > Se requiere agregar, cambiar o eliminar algunos casos en momento de ejecución
- > Se requiere tener consistencia de datos

Estas decisiones pueden ser causa de la creación de una clase que no tiene comportamiento más que simples setters y getters de sus atributos.

Lo importante es entender que:

- > Una ENTIDAD PUEDE QUE NO TENGA COMPORTAMIENTO Y ESO NO LA CONVIERTE EN UN VALUE OBJECT pues tiene sentido de existencia propio y es algo “muy relevante” para nuestro dominio
- > Un value object es un objeto inmutable, desechable, accesorio a algo principal, que no tiene sentido de existencia propio a menos que sea contextualizado
- > Una entidad puede tener comportamiento (lógica de negocio) y atributos y ser una “entidad”

Modelado de Usuarios, roles y permisos (Parte 1)

RESPONSABILIDADES DE CAPAS DE UN SISTEMA

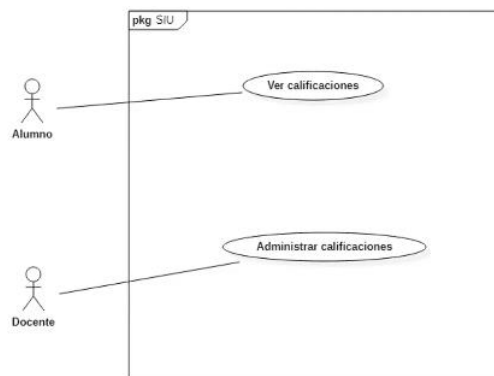
Muchas veces en el Paradigma Orientado a Objetos nos puede confundir la asignación de responsabilidades entre las distintas capas que puede tener un Sistema. Esta confusión se puede ver en la asignación incorrecta de métodos que tienen un nombre de “funcionalidades completas”, de “botones de interfaz de usuario” o de “Casos de Uso” a clases que representan distintos actores.

Resulta que este diseño es incorrecto porque no debemos mezclar la lógica de negocio con los permisos que tiene un actor/rol sobre nuestro Sistema.

Por ejemplo:

El Sistema de Gestión Educativa SIU Guaraní, el diagrama de Casos de Uso y los siguientes requerimientos:

- El sistema debe permitir que los alumnos visualicen sus calificaciones
- El sistema debe permitir la administración (alta, baja y modificación) de calificaciones por parte de los docentes



→ Sería **INCORRECTO** plantear:

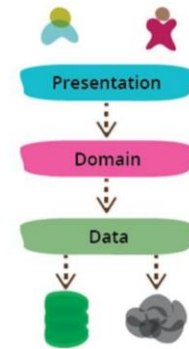
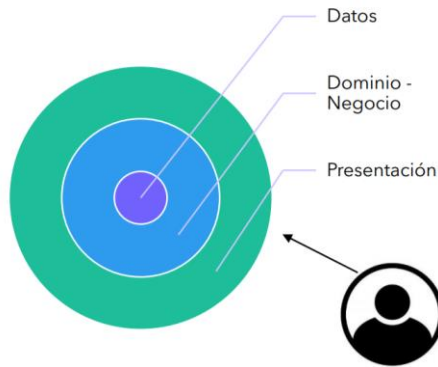
- > Una clase **Alumno** con un método “verCalificaciones”: ¿qué hace este método? Nada, el verbo “ver” me está dando a entender que tendría que “ver algo por pantalla” y el objeto no muestra nada por pantalla
- > Una clase **Docente** con un método “administrarCalificaciones”: ¿qué hace este método?

Entonces, ¿QUÉ ES LO CORRECTO?

La discusión se centra en que no debemos mezclar las responsabilidades de las distintas capas del Sistema.

Pensemos que, si quisiéramos representar todos los casos de uso en sus clases, nos quedaría una clase Dios muy acoplada con las otras clases y lo mismo sucedería con todas las clases de nuestro sistema. Para evitar esto, vamos a distinguir las distintas capas que tiene un sistema y las responsabilidades que tiene cada capa del sistema.

Un Sistema puede dividirse mínimamente en tres capas:



CAPA DE PRESENTACIÓN

El Usuario, ya sea un sistema o persona física, se va a comunicar con nuestro Sistema mediante una **CAPA DE PRESENTACIÓN**.

La **CAPA DE PRESENTACIÓN** es la encargada de presentar los datos al usuario, ya que el usuario es quien interactúa con ella.

La presentación de datos no es solo visual, la presentación puede darse mediante interfaz gráfica o por otros medios. Se pueden distinguir dos formas de presentar los datos:

1. Presentación de datos mediante interfaz gráfica -> interfaz desktop, web, app móvil, etc.
2. Presentación de datos mediante APIs -> API REST

CAPA DE DOMINIO-NEGOCIO

La **CAPA DE DOMINIO-NEGOCIO** es la encargada de modelar las reglas de negocio, las entidades de dominio. Se encarga de modelar las entidades y value objects del dominio.

Contiene la parte estructural y de comportamiento que brinda sostén a todos los posibles Casos de Uso del Sistema.

CAPA DE DATOS

La **CAPA DE DATOS** es la capa encargada de la manipulación de la PERSISTENCIA de los datos del Sistema o la capa encargada del “guardado de los datos”. La capa de Datos se comunica con el medio persistente.

Esta capa es capaz de responder a preguntas como: ¿dónde obtengo X cosa del Sistema?, ¿dónde recupero X cosa del Sistema?, ¿dónde guardo X cosa en mi Sistema? y similares.

Si existe persistencia en una base de datos, esta capa es la encargada de comunicarse con el medio persistente.

RETOMAMOS EL EJEMPLO DEL SIU

Para el caso del SIU sería correcto plantear una clase **Calificación** que tenga como atributo una “nota” numérica, una fecha/hora, curso/asignatura/cuatrimestre/profesor y alumno.

Esta clase **Calificación** pertenece a la capa de DOMINIO-NEGOCIO de nuestro Sistema, ya que en algún momento va a ser instanciada por “alguien” que va a dar de alta nuevas calificaciones.

Pero todavía nos queda algo, ¿cómo vamos a permitir que un docente dé de alta (instancie) una calificación? Y, ¿qué pasa si también tenemos un administrador de plataforma y un bedel que pueden administrar calificaciones? Frente a todas estas preguntas, tenemos que pararnos sobre nuestro dominio, nuestro contexto y los intereses que presenta ese dominio/contexto. ¿Nos interesa la entidad Administrador, Bedel y Docente para nuestro Dominio? La clase **Docente** sí nos interesa, ya que de esta clase podemos realizar otras relaciones que son importantes para el Sistema, el SIU, (asignarle cursos al docente, puntaje, materias, encuestas, etc.), pero con Bedel o Administrador no tenemos ninguna relación importante para nuestro dominio, podríamos solo otorgarle un acceso y nada más.

MODELADO DE USUARIOS, ROLES Y PERMISOS

Llamaremos **USUARIO** a aquella entidad que CONTIENE LOS DATOS DE ACCESO para que una persona física u otro Sistema pueda identificarse en nuestro Sistema al usarlo. La clase USUARIO tiene los datos de acceso al Sistema.

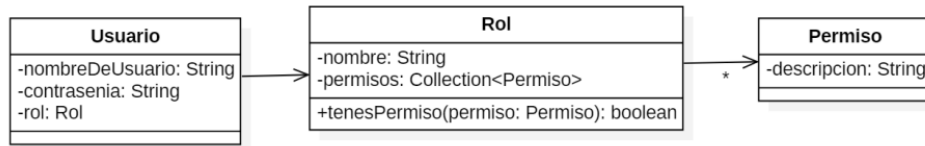
Los DATOS DE ACCESO están compuestos por nombre de usuario y contraseña.

Una misma persona física o Sistema podría cumplir varios roles dentro de nuestro Sistema y ejecutar diferentes casos de uso.

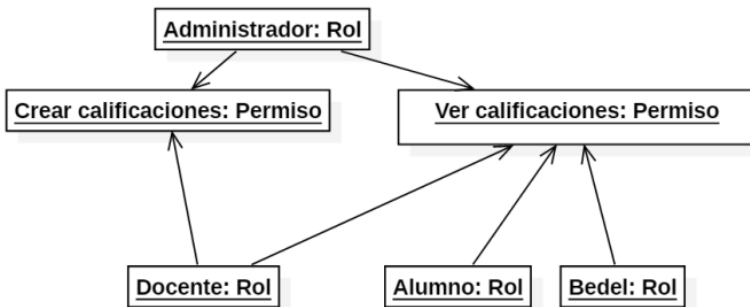
Distintos roles podrían tener la capacidad de ejecutar el mismo caso de uso:

- >Cada rol podría ejecutar muchos casos de uso;
- >Para ejecutar un caso de uso se necesita tener uno o varios permisos, es decir puede ser un permiso para ejecutar un solo caso de uso o un conjunto de permisos para ejecutar un solo caso de uso;
 - Ejemplo del SIU: un “permiso para dar de alta una calificación”, “permiso para modificar una calificación”, etc.
- >Un mismo permiso podría ser otorgado a varios roles distintos.

*Una propuesta, si consideramos que un usuario pueda tener SOLO UN ROL, puede ser:



EJEMPLO DE INSTANCIAS SOBRE SIU - (Diagrama de Objetos)



El **Docente** tiene un permiso, en su colección de Permisos, de “crear calificaciones” y de “ver calificaciones”; y así con todos los otros.

→ No vamos a tener una instancia de Rol por cada Usuario, reutilizamos el objeto Rol.

Esta no es la única propuesta para los Usuarios, roles y permisos:

- > Los permisos podrían estar modelados con enumerados, entendiendo que esto trae ciertas limitaciones
- > Los usuarios podrían tener múltiples roles (colección de roles)

Ahora, ¿cómo vamos a relacionar la entidad Docente con la entidad Usuario?

Para crear la relación entre la entidad **Docente** y la entidad **Usuario**, es decir que un Docente tenga sus datos de acceso al Sistema, dentro de la entidad Docente vamos a tener un atributo Usuario. De la misma forma, un **Alumno** va a tener un atributo **Usuario**.

Pero ¿qué pasa si un Alumno puede ser Docente al mismo tiempo? (de asignaturas que ya cursó)

En caso de que un **Alumno** sea también **Docente** vamos a tener una instancia de la clase **Alumno** y una instancia de la clase **Docente** que van a representar a una misma persona, por lo tanto, ambas clases van a tener que referenciar a la misma instancia de **Usuario**, es decir que ambas clases van a estar “apuntando” a la misma clase Usuario.


Todas estas cosas que mencionamos sobre los permisos, instanciar objetos y demás, van a reflejarse en la **CAPA DE CONTROLADORES**.

CAPA DE CONTROLADORES

La **CAPA DE CONTROLADORES** es la capa que está arriba de la capa de Dominio-Negocio, se encuentra entre esa capa y la de Presentación.

En esta capa se suelen agregar los métodos que orquestan las funcionalidades completas/el caso de uso, además de verificar que los usuarios que están tratando de realizar ciertas acciones tengan los permisos necesarios.

En algunos Sistemas se delega la verificación de los Permisos en una capa llamada MIDDLEWARE.

IDAÑEZ, LUCIA MARÍA 

RETOMANDO EL EJEMPLO DEL SIU EN CÓDIGO

```
public class CalificacionesController {
```

```
    public void crearCalificacion(DataCalificacion data, Usuario usuario) {
        Permiso permisoCrearCalificaciones = RepositorioDePermisos.buscar("CREAR_CALIFICACIONES");

        if(!usuario.getRol().tenesPermiso(permisoCrearCalificaciones)) {
            throw new PermisoInsuficienteException(permisoCrearCalificaciones);
        }

        Calificacion unaCalificacion = new Calificacion();
        //...

        RepositorioDeCalificaciones.guardar(unaCalificacion);
    }
}
```

DataCalificacion podría ser un value object, ya que tiene los datos que ingresó

Usuario es el usuario logueado

En la primera línea tenemos:

```
Permiso permisoCrearCalificaciones = RepositorioDePermisos.buscar("CREAR_CALIFICACIONES");
```

Donde estamos llamando a un **RepositorioDePermisos** y le estamos pidiendo que **busque** la instancia de "CREAR_CALIFICACIONES", entonces el repositorio busca esa instancia el medio persistente (el repositorio está en la capa de datos, por lo tanto, el controlador interactúa con la capa de datos) y la almacena en **permisoCrearCalificaciones**.

```
if(!usuario.getRol().tenesPermiso(permisoCrearCalificaciones)) {
    throw new PermisoInsuficienteException(permisoCrearCalificaciones);
}
```

En este caso chequeamos que, si el **Rol** del usuario **NO** tiene el **permisoCrearCalificaciones**, se lance una Excepción. Esa Excepción lanzada se debería agarrar desde una capa de Presentación para comunicarle al cliente que no puede realizar esa acción porque no tiene el permiso suficiente.

```
Calificacion unaCalificacion = new Calificacion();
//... Se configura la Calificación
```

```
RepositorioDeCalificaciones.guardar(unaCalificacion);
```

En caso de que tenga permiso, se instancia la **Calificación** y se guarda en el **RepositorioDeCalificaciones**.

*En ningún momento hacemos algo así como **docente.crearCalificación()** porque no nos importa, lo único que nos interesa es que el **ROL DEL USUARIO QUE INTENTA CREAR LA CALIFICACIÓN TENGA LOS PERMISOS NECESARIOS**.

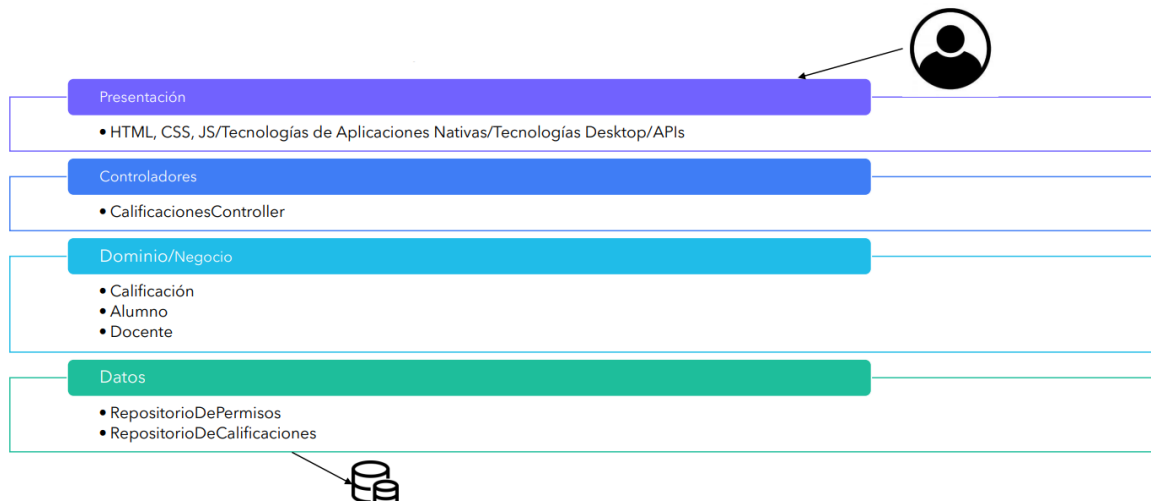
Cosas para tener en cuenta:

→ Los repositorios son los objetos encargados de la persistencia de los demás objetos (guardar, buscar, eliminar, entre otros)

→ La instanciación y configuración de la calificación podría realizarse mediante la ayuda de algún patrón creacional

→ Los controladores son los únicos que deberían comunicarse con las demás capas. Los controladores van a jugar con todo y orquestan la solución completa

*Ubicando geográficamente a nuestros objetos en las distintas capas:



IDAÑEZ, LUCIA MARÍA 

Biblioteca vs. Framework

BIBLIOTECA

La **BIBLIOTECA** se desarrolla, se compila y se carga de manera separada al código de nuestro sistema, sin embargo, sus funcionalidades son usadas indistintamente por nuestro código como si la biblioteca fuera parte de él.

Las bibliotecas:

- ✓ resuelven problemas de reutilización de lógica asociada a abstracciones;
- ✓ pueden ser propias o de terceros;
- ✓ definen funcionalidades concretas y estructuras que pueden ser utilizadas o no por el desarrollador en algún momento.

El control del flujo del programa lo tiene el desarrollador, él llama a la biblioteca para que resuelva algo, la biblioteca lo resuelve y el flujo vuelve al desarrollador. Entonces, nosotros llamamos a la biblioteca cuando la necesitamos, ella no nos llama a nosotros.

En las bibliotecas, es responsabilidad del desarrollador decidir cómo y cuándo se va a utilizar los componentes.

Las decisiones de diseño que se toman suelen tener bajo impacto en el diseño del código.

Utilizan **control directo**, es decir que el usuario llama funciones de la biblioteca e instancia las estructuras de la biblioteca a definir.

FRAMEWORK

Los **FRAMEWORKS** definen una estructura, una forma de trabajar siguiendo una serie de reglas. En los frameworks no solo se definen funciones y componentes que pueden reutilizarse, sino que también se definen piezas de software reutilizables que pueden ser extendidas para cubrir necesidades, es un marco de trabajo común para todos estos elementos.

Los Frameworks condicionan el diseño de un componente de software. Suelen manejar ellos el flujo de ejecución del programa e invertir el control, son ellos quienes deciden cuándo llamar al código del usuario. El desarrollador nunca define un método main sino que el framework lo define y en algún momento llama al código del usuario.

El Framework define la forma en la que se estructura el código y el desarrollador se ve obligado a seguir esa forma.

Las decisiones de diseño que se toman pueden condicionar mucho el diseño del código del cliente.

Utilizan **control inverso**, es decir que el framework llama funciones abstractas que el usuario define en concreto.

Tipos de Framework:

Según la rigidez, se clasifican en dogmáticos o no dogmáticos:

DOGMÁTICO: son aquellos frameworks que tienen ciertas restricciones, ofrecen un soporte para el desarrollo rápido en un dominio particular porque la manera correcta de hacer cualquier cosa está bien comprendida y documentada.

NO DOGMÁTICO: son aquellos frameworks que no tienen tantas restricciones sobre el mejor modo de unir componentes para alcanzar un objetivo o incluso qué componentes deben usarse. Hace más fácil para los desarrolladores usar las herramientas adecuadas para completar una tarea.

Inyección de Dependencias

Partimos de un ejemplo:

>> **Incidente**

```
public void enviarAvisos(){
    this.interesados.forEach(i -> whatsappSender.enviar(i, "Ha ocurrido un incidente"));
}
```

Suponiendo que existe un objeto **whatsappSender**, ¿cómo llegamos a conocerlo? Algunas opciones:

- Tener una única instancia de su clase accesible globalmente
- Obtener ese objeto a través de otro que lo provea
- Que a la clase Incidente le llegue por parámetro whatsappSender

PATRÓN SINGLETON

>> Incidente

```
public void enviarAvisos(){
    this.interesados.forEach(i -> WhatsAppSender.getInstance() enviar(i, "Ha ocurrido un
    incidente"));
}
```

Devuelve la única instancia

GLOBAL DE LA CLASE

```
>> WhatsAppSender
private static WhatsAppSender instance = null;

public static WhatsAppSender getInstance() {
    if(instance == null)
        instance = new WhatsAppSender(); // más toda la configuración
    return instance;
}
```

El **PATRÓN SINGLETON** es un PATRÓN CREACIONAL.

Este patrón asegura que exista una ÚNICA INSTANCIA DE UNA CLASE.

Para que sea un objeto Singleton y siempre devuelva la única instancia global, debe tener lo siguiente:

```
>> WhatsAppSender
private static WhatsAppSender instance = null;
```

```
public static WhatsAppSender getInstance() {
    if(instance == null)
        instance = new WhatsAppSender(); // más toda la configuración
    return instance;
}
```

La **primera vez** que se llama a getInstance, se pregunta si ese atributo "instance" es nulo, en caso de serlo, se instancia con new WhatsAppSender() y se devuelve esa instancia (ese instance)

La **segunda vez** que se llame, instance no va a ser null entonces se devuelve siempre la misma instancia de WhatsAppSender

Lo único que falta para que esto sea un Singleton es que el constructor de WhatsAppSender sea "private".

SERVICE LOCATOR

>> Incidente

```
public void enviarAvisos(){
    this.interesados.forEach(i -> ServiceLocator.get("whatsappSender").enviar(i, "Ha ocurrido
    un incidente"));
}
```

Llamamos a una clase ServiceLocator con el método **get** y le pasamos "whatsappSender", esto nos va a devolver una instancia de la clase WhatsAppSender, el cómo nos devuelve esa instancia no nos importa.

INYECCIÓN DE DEPENDENCIAS

>> Incidente

```
public Incidente(WhatsAppSender whatsappSender){...}

public void enviarAvisos(){
    this.interesados.forEach(i -> this.whatsappSender.enviar(i, "Ha ocurrido un incidente"));
}
```

A la clase **Incidente** es su constructor le llega por parámetro una instancia de WhatsAppSender y lo único que hace es llamar a su atributo en el método de **enviarAvisos()**.

El quién le provee esa instancia que le llega como parámetro no nos interesa, eso es trabajo de la inyección de dependencias.

Suponiendo que **A**, instancia de la clase 1, necesita a **B**, instancia de la clase 2, para poder realizar la tarea **X**:

Singleton	Service Locator	Inyección de Dependencias
<ul style="list-style-type: none"> • A solicita a clase 2 una instancia, la cual devuelve siempre la instancia B. • "B" es un objeto global, única instancia para toda la ejecución. • Fuerte acoplamiento entre Clase 1 y Clase 2. • Difícil de testear pues es complicado mockear a B 	<ul style="list-style-type: none"> • A solicita al Service Locator alguien que sea capaz de realizar la tarea X y éste le devuelve la instancia B o algún otro objeto que cumpla con la misma interface. • El Service Locator es un objeto global que permite generar distintas configuraciones. • Permite el mockeo de objetos 	<ul style="list-style-type: none"> • A recibe como parámetro, en su constructor, a B o algún otro objeto que cumpla con la misma interface. • A no solicita a nadie la instancia B (o algún otro similar), sino que "le llega desde afuera". • Es más testeable pues permite el mockeo de objetos. • Se puede combinar con las anteriores.

Clase 17/05

Arquitectura de Software

La **ARQUITECTURA DE SOFTWARE** es la estructura o estructuras del sistema que está formada por **componentes de software**, las propiedades externas visibles de esos componentes, es decir una forma de comunicarnos con ellos, y las relaciones entre ellos.

Un **COMPONENTE** es una PIEZA DE SOFTWARE que puede ser código fuente, código binario, un ejecutable, o algo con una interfaz definida, una forma de comunicación con esos componentes.

¿Qué es la Interfaz de un Componente?

Una INTERFAZ establece las operaciones externas de un componente, estas operaciones determinan una parte del comportamiento del componente.

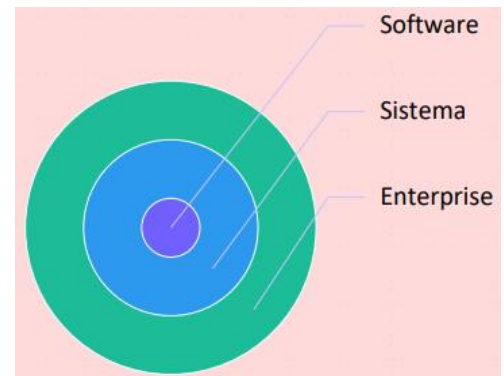
¿Cómo se relacionan con otros componentes? (En UML)

Las relaciones entre componentes se representan mediante dependencias, es decir las DEPENDENCIAS pueden ser entre componentes o entre un componente y la interfaz de otro, es decir uno de ellos usa los servicios o funcionalidades del otro.

NIVELES

La arquitectura de software tiene muchos NIVELES, podemos hablar de arquitectura de software, arquitectura de sistema o arquitectura Enterprise.

- ✓ La **arquitectura de software** es aquella arquitectura para una aplicación o subsistema en particular.
- ✓ La **arquitectura de sistema** se conforma por la arquitectura de software y la infraestructura, es decir donde se despliega y funciona ese sistema.
- ✓ La **arquitectura Enterprise** define la estrategia tecnológica y de negocio de la organización para el desarrollo de los sistemas, es la arquitectura que adopta toda organización. Condiciona a todos los sistemas que se van a estar creando dentro de la organización.
->Las restricciones que tiene la arquitectura Enterprise son las reglas de negocio de la organización



CARACTERÍSTICAS

- La arquitectura de Sistema es diseño de alto nivel, es diseño de nivel estratégico. Las decisiones que tomamos son de alto impacto.
- No es solo lógico sino también físico y organizacional, la arquitectura de sistema abarca la parte lógica y la parte de infraestructura.
- La arquitectura contiene las estrategias para resolver los atributos de calidad.
- Define el Sistema en términos de componentes y de interacción entre ellos.
- Todo lo que decidamos, no lo vamos a poder cambiar fácilmente, ya que es una decisión arquitectónica. Por esto mismo, la arquitectura de sistema gobierna al sistema por mucho tiempo.

IDAÑEZ, LUCIA MARÍA 

DECISIONES ARQUITECTÓNICAS

- Las decisiones arquitectónicas afectan a muchas partes del sistema y son difíciles de cambiar, ya que, al ser decisiones de alto nivel, todo lo que esté “debajo” se ve afectado.
- Las decisiones arquitectónicas dan estructura al sistema.
- Identifican y reducen riesgos.
- Las decisiones arquitectónicas deben tomarse en forma temprana, ya que, si las tomo más tarde, corro el riesgo de que sea muy tarde para cambiar algo.

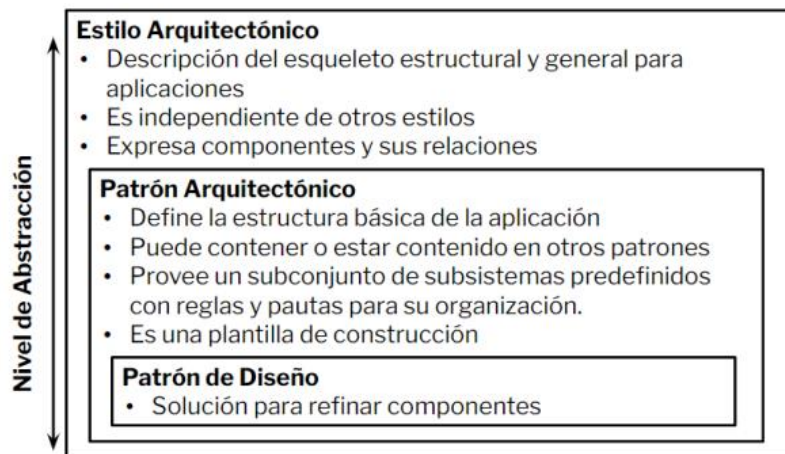
ENTRADAS DE LA ARQUITECTURA DE SOFTWARE

- Requerimientos funcionales
- Requerimientos no funcionales y atributos de calidad
- Restricciones de negocio
- Restricciones técnicas -> parte Enterprise
- Futuros requerimientos
- Experiencia del Arquitecto
- Estilos y patrones arquitectónicos

Los **REQUISITOS FUNCIONALES** son aquellos que responden a la necesidad del usuario.

Los **REQUISITOS NO FUNCIONALES** son aquellos que corresponden a las propiedades del sistema (rendimiento, seguridad, etc.)

NIVELES DE ABSTRACCIÓN

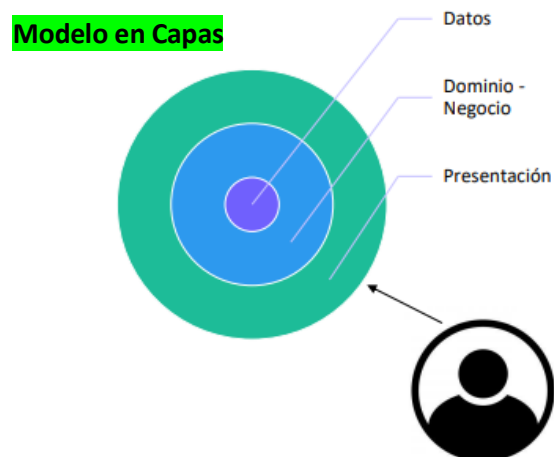


Modelo en Capas

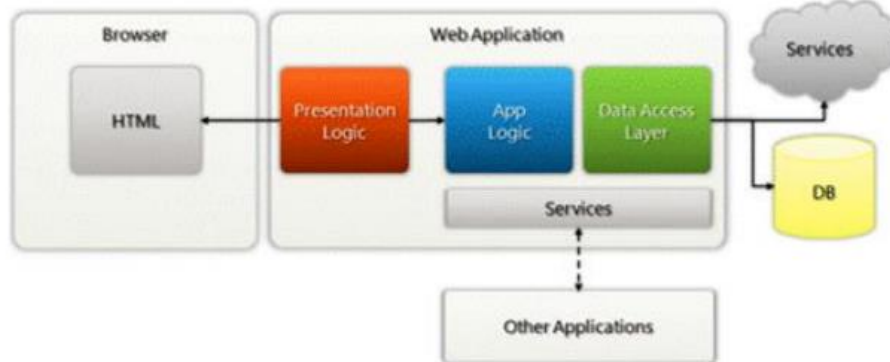
El **MODELO EN CAPAS** es una técnica que consiste en dividir en capas un problema de diseño complejo.

Al dividir el problema en capas, asignamos a cada capa una responsabilidad, esto quiere decir que cada una tiene un rol único en el sistema, de esta manera mejoramos la cohesión.

Las capas superiores usan los servicios de las inferiores, pero las capas inferiores no usan los servicios de las superiores.



ARQUITECTURA CON LA QUE VAMOS A ESTAR TRABAJANDO EN EL TP



En esta arquitectura tenemos dos componentes importantes, un Browser y un Web Application.

El Browser va a estar corriendo en un equipo cliente (un navegador), dentro del browser tenemos el código HTML.

El Web Application está corriendo en el servidor, dentro del Web Application vamos a tener ejecutando mi componente y este componente está formado por capas, como la capa de presentación, la capa lógica o de dominio y la capa de acceso a datos. Hay una capa anexa que se llama Capa de Servicio, la cual se comunica con otras aplicaciones; es saliente la comunicación, no entrante.

VENTAJAS

- Nos mantiene enfocados en un solo problema a resolver, cada capa tiene un solo problema a resolver.
- Esconde el detalle de cómo se lleva a cabo aquellos servicios que expone; cada capa está totalmente encapsulada, solo conozco la interfaz que brinda esa capa.
- Las capas deberían ser reemplazables, dado que el acoplamiento entre ellas es muy bajo.
- Minimiza la dependencia entre componentes, cada capa depende de la inmediata superior e inferior. Para “arriba”, brindamos servicios mientras que, para “abajo”, consumimos servicios.
- Facilita ciertas pruebas porque podemos ir probando por separado cada capa.

DESVENTAJAS

- Los cambios pueden generar efecto cascada; si agregamos algo en una capa, debemos agregarlo también en las capas que están por debajo.
- Muchas capas agregan complejidad y afectan de manera negativa al rendimiento.

Cliente – Servidor

El **CLIENTE – SERVIDOR** tiene dos grandes componentes: el cliente y el servidor.

→ Un **SERVIDOR** provee uno o más servicios a través de una interfaz.

→ Un **CLIENTE** usa los servicios que provee el Servidor como parte de su operación en el acceso al servidor

CLASIFICACIÓN SEGÚN RESPONSABILIDADES

Una arquitectura Cliente – Servidor se puede clasificar dependiendo las responsabilidades asignadas a sus componentes.

Cientes Activo, Servidor Pasivo

El **cliente** es quien posee la mayor lógica de negocio mientras que el **servidor** limita sus funcionalidades a la persistencia, esto quiere decir que el servidor solo nos brinda los datos.

Cliente Pasivo, Servidor Pasivo

Ambos componentes, tanto el **cliente** como el **servidor**, poseen poca lógica de negocio o son considerados “componentes intermedios” de algo más grande.

Cliente Pasivo, Servidor Activo

Conocido como “cliente liviano”. El **servidor** posee toda lógica negocio mientras que el **cliente** solo brinda los datos.

Esta es la arquitectura web tradicional.

Cliente Activo, Servidor Activo

Conocido como “cliente pesado”. La lógica de negocio está del lado del **cliente** y del lado del **servidor**, el cliente posee la lógica de presentación de datos. El cliente tiene mucha responsabilidad, es decir hay mucha lógica corriendo del lado del navegador, y también hay mucha lógica corriendo del lado del servidor.

VENTAJAS Y DESVENTAJAS DE CLIENTE PASIVO – SERVIDOR ACTIVO

VENTAJAS

Mantenibilidad: cambios de funcionalidad centralizados

Seguridad: Centralización de control de acceso a recursos

DESVENTAJAS

Eficiencia (tiempo de respuesta): el servidor puede ser un cuello de botella

Disponibilidad: único punto de falla

MVC – Patrón Arquitectónico

Modelo – Vista – Controlador (MVC) es un patrón de arquitectura de software DE INTERACCIÓN, esto quiere decir que es un patrón arquitectónico que está pensado para la interacción con un usuario físico por medio de una interfaz gráfica.

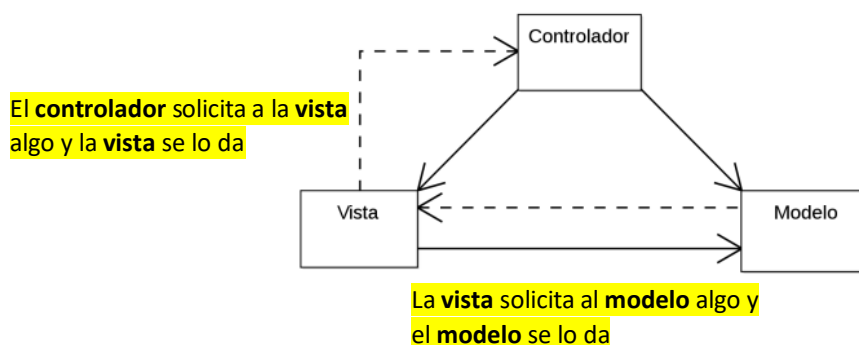
El MVC separa la arquitectura en tres componentes: el modelo, la vista y el controlador.

El controlador va a orquestar casi todo el sistema, va a responder a todos los casos de uso, se va a comunicar con la vista y el modelo.

La vista conoce el modelo

El modelo no tiene que conocer que existen controlador y la vista tampoco debe conocer que existen los controladores.

Dentro del modelo podemos diferenciar los repositorios y las entidades. Los repositorios son esos objetos que se encargan de la persistencia y se comunican con una base de datos.



MODELO

El **MODELO** representa los datos con los cuales va a trabajar el Sistema.

- Encapsula el estado del sistema
- Gestiona todos los accesos a dichos datos (consultas, actualizaciones, etc.)
- Valida la especificación de la lógica detallada en el “negocio”
- Envía a la Vista, por medio del Controller, datos para que sean visualizados
- El Modelo se puede dividir en Entidades, las cuales representan la Capa de Dominio – Negocio, y Repositorios que pertenecen a la Capa de Datos

CONTROLADOR

El **CONTROLADOR** responde a las acciones del usuario y realiza peticiones al modelo cuando se hace alguna solicitud de datos.

- Es el intermediario entre el Modelo y la Vista
- Define el comportamiento del Sistema
- Traduce las acciones del usuario en actualizaciones del modelo
- Es el gestor del ciclo de vida del sistema
- Responde a las acciones del usuario (o eventos)
- Realiza peticiones al modelo

IDAÑEZ, LUCIA MARÍA 🐱

El Controlador es quien responde a la solución general, quien realiza solución.

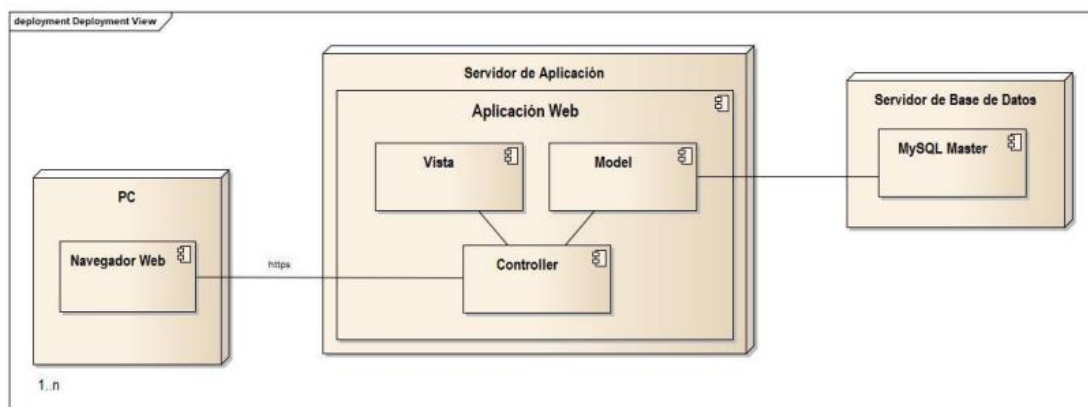
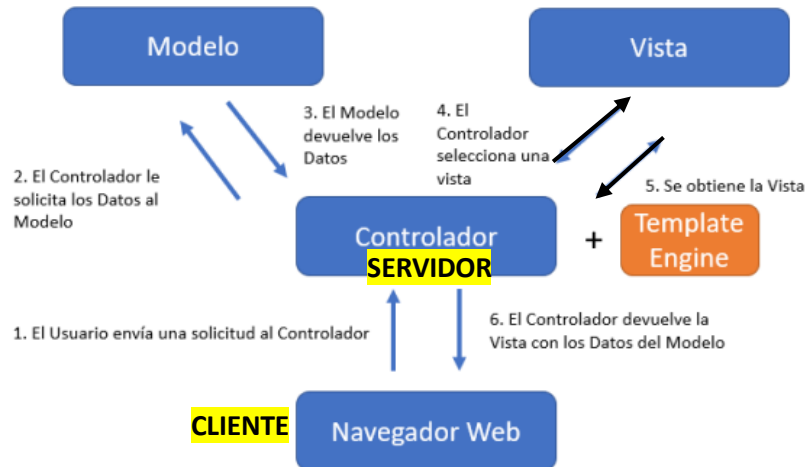
Los **DTO o Data Transfer Object** son Value Objects que agarran los datos que envió el Usuario y los vamos a usar para instanciar nuestros objetos de negocios

VISTA

La **VISTA** presenta los datos y la forma de interactuar con ellos en un formato adecuado para el usuario.

- Posee la lógica que permite poder presentar los datos de forma amigable para el usuario
- Envía las acciones del usuario al controlador
- Solicita actualizaciones al modelo a través del controlador
- La comunicación entre la vista y el controlador se realiza mediante la transferencia de DTO's.

MVC – WEB



VALIDACIÓN DE DATOS

¿Dónde se validan los datos?

- Los datos se validan EN CADA CAPA. Esto se denomina **validación profunda**.
- En la vista se pueden validar los campos requeridos y consistencia (números y/o letras donde corresponda, entre otras cosas).
- En el controlador se debería volver a validar la consistencia de los datos y los campos requeridos. También se puede realizar una validación con un servicio externo.
- En el modelo se deberían realizar validaciones de negocio.

VENTAJAS Y DESVENTAJAS DE MVC

VENTAJAS

Buena separación de las responsabilidades, tenemos la lógica de las vistas, los controladores y los modelos por separado

Reusamos las vistas y controladores

Flexibilidad

DESVENTAJAS

Mayor complejidad de los sistemas

No siempre útil en aplicaciones con “poca interacción” o con “vistas simples”

Difícil de testear como un todo

IDAÑEZ, LUCIA MARÍA 

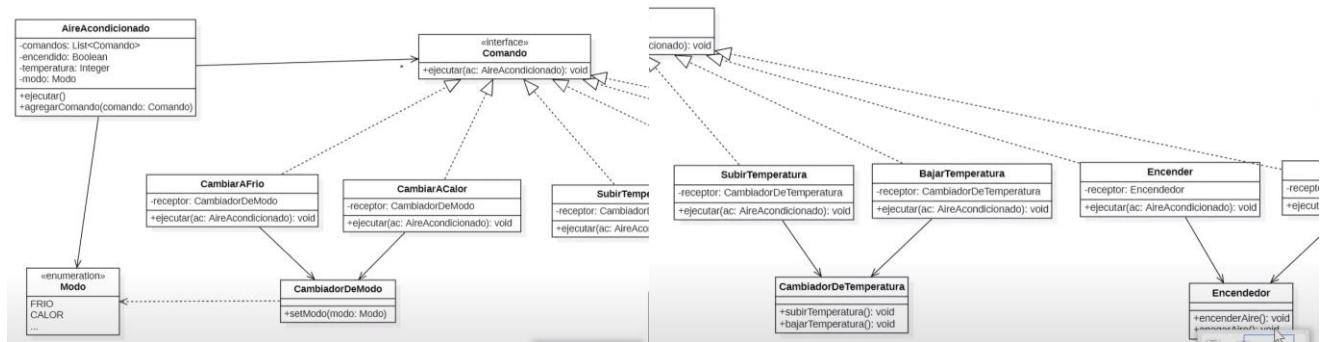
Patrón Command

PATRÓN COMMAND es un patrón de COMPORTAMIENTO.

Motivación: Se solicitó el diseño de un aplicativo para dispositivos móviles que pueda controlar a distancia el aire acondicionado. Para la primera versión se solicitaron los siguientes comandos:

- >Prender y apagar
- >Subir y bajar temperatura
- >Cambiar modo

DIAGRAMA DE CLASES:



En todos los comandos vamos a tener el receptor y nuestro método ejecutar que viene de la interfaz, el método ejecutar lo que hace es llamar al método que realiza la acción concretamente del receptor.

En la clase encender a través del atributo **receptor:Encendedor** va a llamar al método **encenderAire()** que tiene el atributo **Encendedor**, la clase **BajarTemperatura** en su método **ejecutar(AireAcondicionado)** va a bajar la temperatura del aire en un grado y para esto va a llamar al método **bajarTemperatura()** que posee el atributo **receptor:CambiadorDeTemperatura**. Entonces, quien realmente ejecuta la acción son nuestros receptores mientras que nosotros encapsulamos ese comportamiento en **CambiarAFrio**, **CambiarACalor**, etc.

En nuestra clase **AireAcondicionado**, vamos a agregar un Comando a la lista de Comando y esa lista de comandos es la que se va a ejecutar cuando nosotros llamemos al método **ejecutar()** en la clase **AireAcondicionado**.

¿Qué hace?

- >Estructuras complejas que no están directamente acopladas al dominio
- >Acciones que sabemos que se van a ejecutar, pero no sabemos quién las va a ejecutar
- >Genera una abstracción por cada acción que se quiere realizar, encapsula cada comportamiento en una clase

→ Cada acción que se quiere realizar es un Comando, las clases tienen nombres de métodos

- >Permite enviar órdenes para que alguien las realice en ese momento o en un momento posterior, sin interesarnos quien recibe estas órdenes

Se sugiere su utilización cuando:

- >Se requiere configurar en momento de ejecución una serie de acciones que debe realizar un objeto
- >Se requiere configurar en momento de ejecución una serie de acciones que debe realizar el objeto en un momento posterior al de configuración, formando una cola.
- >Se requiere seguir adelante con el diseño e implementación sabiendo qué se quiere hacer, pero sin saber exactamente cómo hacerlo o quién lo hará.
- Hace referencia que al estar implementado con una interfaz Comando la cual está atada únicamente a la clase AireAcondicionado, nos permite crear nuevos Comandos fácilmente, ya que sabemos en nuestra clase AireAcondicionado se van a ejecutar sin problemas dado que los va a tratar a todos polimórficamente (gracias a la interfaz)

COMPONENTE:

→ **Invocador**: Clase concreta que utiliza a los comandos concretos de forma polimórfica, sin saber los receptores de estos comandos o qué hacen. (En nuestro caso era AireAcondicionado)

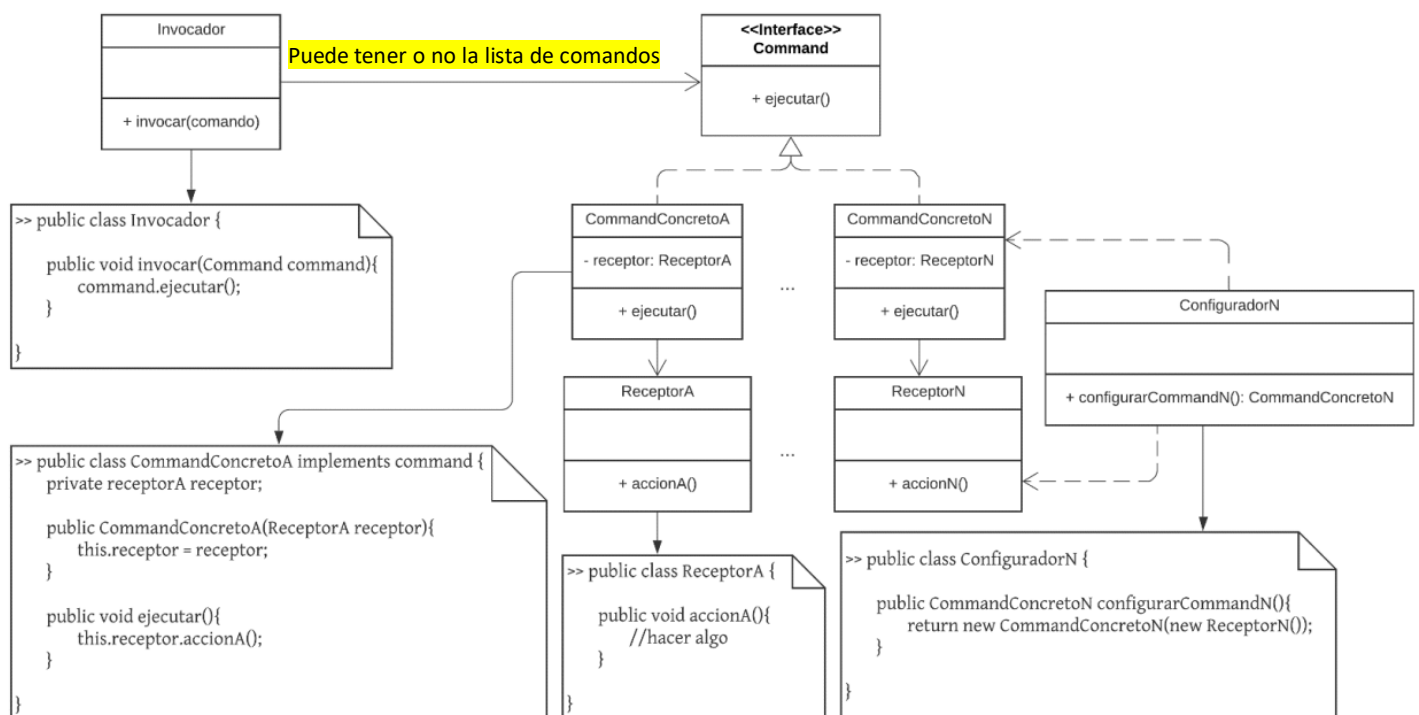
→ **Interface (o clase abstracta) Command**: Interface que define la firma que tendrán todos los comandos concretos. Su propósito es que el Invocador esté acoplado a la clase Command, de esta forma la clase Invocador puede estar totalmente desacoplada de los comandos concretos.

→ **Comandos concretos**: Clases concretas que implementan la interface Command, estas clases encapsulan la utilización de Receptores. Los comandos concretos suelen ser los orquestadores de una acción ya que a quien realmente realiza la acción (los Receptores) y en un orden particular.

→ **Receptores**: Clases que contienen la implementación/lógica de la acción que se va a realizar, son quienes hacen todo el trabajo.

Ventajas del patrón Command:

- ✓ Mayor mantenibilidad debido a que el comportamiento de las acciones es fácilmente localizable
- ✓ Alta cohesión en los comandos concretos, ya que realizan una sola acción (la cual es llamar a los receptores)
- ✓ Fácil extensibilidad y facilidad para agregar nuevos comandos



Clase 31/05

Esquemas

PULL BASED → Vamos a buscar el dato constantemente

PUSH BASED → Esperamos que el dato nos llegue

Notificaciones

→ Push based, las notificaciones van al sistema, el sistema no va a las notificaciones

Clase 14/06

Patrón Singleton

PATRÓN SINGLETON es un patrón CREADOR.

¿Qué hace?

→ Asegura que existirá una única instancia de una clase

Se sugiere su utilización cuando:

→ Se requiere tener una única instancia de una clase y que la misma sea accesible por terceros

COMPONENTE:

→ **Clase Singleton**: clase que permite que sea instanciada una única vez a través de un método de clase (static). El constructor debe ser privado para que solamente sea accesible por la misma clase.

→ Método **getInstancia()** → Nos permite obtener la instancia

¡CUIDADO!

IDAÑEZ, LUCIA MARÍA 🐱

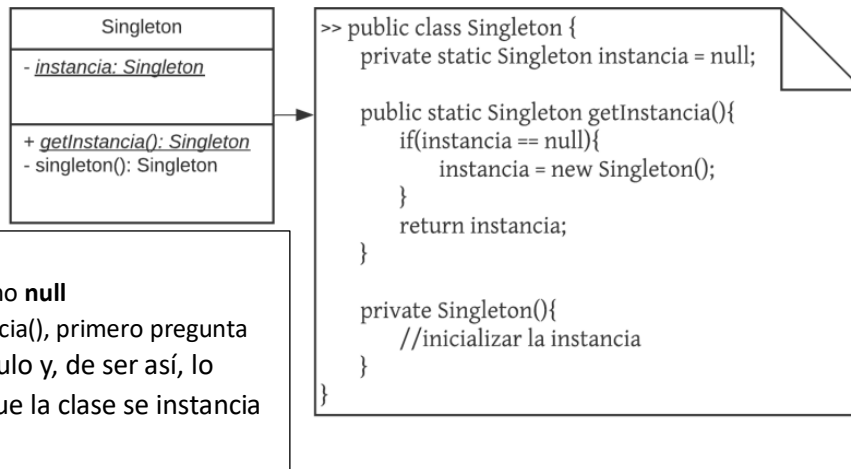
->Este patrón tiene mucho más sentido en lenguajes compilados

->Si bien en lenguajes interpretados también es posible su implementación, se debe tener en cuenta que el objeto va a "morir" luego de cada llamada que se realice al script principal.

WKO ≠ SINGLETON

¿Qué proporciona su uso?

->Evita repetir extensas configuraciones de inicialización de instancias.



Patrón Factory Method

PATRÓN FACTORY METHOD es un patrón CREACIONAL.

¿Qué hace?

->Encapsula en un único punto (**un método**) la creación de uno o varios objetos que tienen una interfaz en común

Se sugiere su utilización cuando:

->Se requiere instanciar una clase de entre varias (con al menos una interfaz en común) según el resultado de alguna condición o según el valor de una variable, en varias partes del proyecto.

->Intenta llevar la instanciación de esa clase a un único método

COMPONENTE:

→ Método Factory: Método de alguna clase que tiene la responsabilidad de instanciar una clase, entre muchas, según algún parámetro/condición. Este método Factory suele ser un método de clase (static/estático). Evita que otras clases tengan lógica repetida de instanciación.

→ Interface (o clase abstracta) Producto: es la interface que comparten las clases que necesitan ser instanciadas de forma sencilla/transparente para un tercero. Esto nos permite tratar indistintamente a todas las clases/Productos concretos.

→ Productos concretos: Clases concretas que implementan (o, si fuera una clase abstracta, heredan) la interface Producto.

→ Cliente(s): Clase(s) cuyos objetos hacen uso del método de fabricación.

Patrón Simple Factory

PATRÓN SIMPLE FACTORY es un patrón CREACIONAL.

¿Qué hace?

->Encapsula en un único punto (**una clase**) la creación de uno o varios objetos que tienen una interfaz en común

ES UNA GENERALIZACIÓN DEL PATRÓN FACTORY METHOD

En el caso del Patrón Factory Method, habíamos dicho que encapsula en un método la creación de uno o varios objetos, pero no dijimos EN QUÉ CLASE LO HACE. Para este caso vamos a crear una clase específica que se encargue de instanciar alguna otra clase.

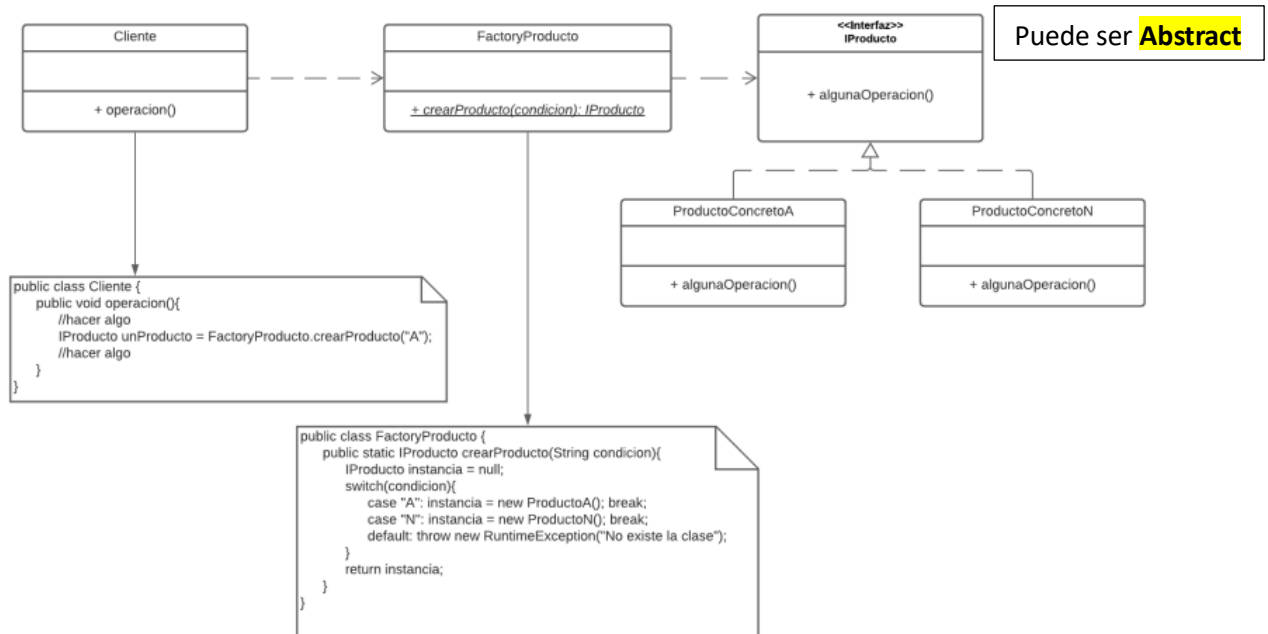
Se sugiere su utilización cuando:

->Se requiere instanciar una clase distinta (con al menos una interfaz común) según el resultado de alguna condición o según el valor de una variable, en varias partes del proyecto; y dichas clases pueden ser consideradas “complejas” de instanciar o necesitan “mucha configuración” para funcionar.

COMPONENTE:

Son los mismos que tiene el patrón Factory method, a excepción de un componente más:

→ **Clase Factory:** Clase concreta que tiene la responsabilidad de instanciar una clase, entre muchas, según algún parámetro y/o condición compleja. Esta clase puede tener varios métodos (internos/privados) que colaboren en la creación de los diferentes objetos. Esta clase contiene al **Factory Method**.



Clase 21/06

Patrón Observer

PATRÓN OBSERVER es un patrón COMPORTAMIENTO.

¿Qué hace?

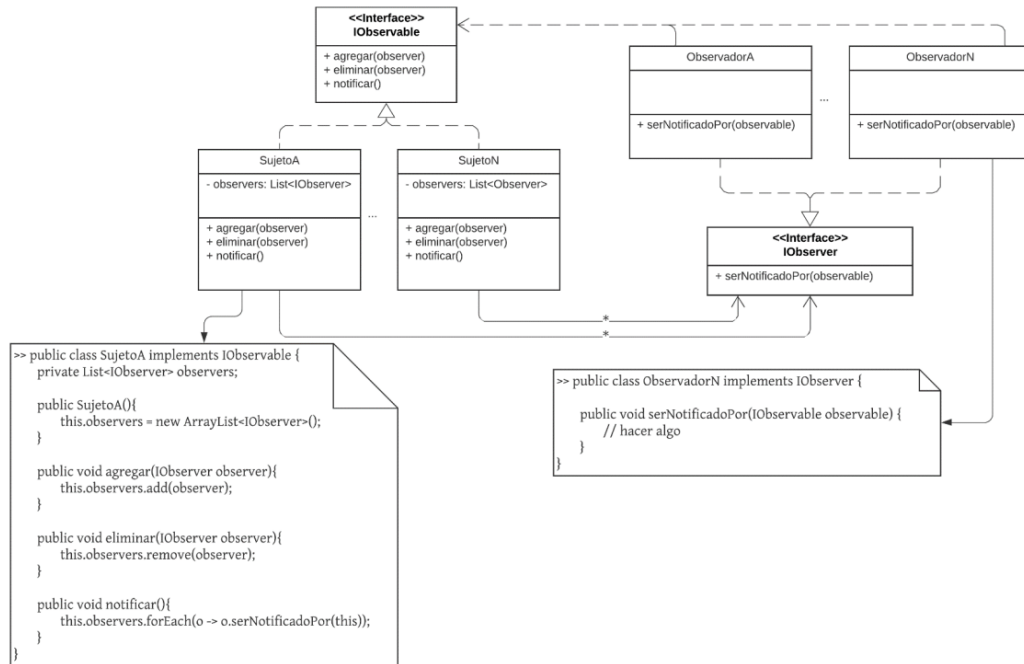
->Permite notificar a aquellos interesados de cierto sujeto sobre los eventos que le ocurren a este sujeto y que ellos actúen sobre esto.

->Desacopla las acciones que se ejecutan de los eventos que las activan (desacopla la notificación del evento que dispara esa notificación).

Se sugiere su utilización cuando:

->Se requiere disparar acciones de diferentes naturalezas frente a la ocurrencia de un evento en algún objeto

->Se requiere notificar/avisar a algunos interesados de un objeto sobre la ocurrencia de algún evento y no interesa saber quiénes son ni cuántos son



Clase 28/06

Patrón Builder

PATRÓN BUILDER es un patrón CREACIONAL.

¿Qué hace?

- >Separa la lógica de la construcción de un objeto.
- >Encapsula el proceso de instanciación de un objeto.
- >Permite modelar reglas de negocio para una determinada entidad.
 - Encapsula la configuración de un objeto y realiza las validaciones necesarias para que se cumplan las reglas de negocio correspondientes a la configuración de ese objeto

Se sugiere su utilización cuando:

- >El objeto que se quiere crear necesita tener muchos atributos configurados antes de que se lo utilice. El objeto no puede ser inconsistente.
- >Se necesitan modelar ciertas reglas de negocio vinculadas a la consistencia de una entidad/objeto.
- >El objeto que se quiere crear es complejo y requiere de “pasos” para poder instanciarse.

