

# Final Programación 2 Repaso

## ¿Qué es un mensaje?

Un **mensaje** es una solicitud para llamar a un método de un objeto con sus respectivos parámetros. Un mensaje es la forma con la que se comunican los objetos.

## ¿Qué es una clase?

Una **clase**

- Tiene nombre, atributos y métodos.
- Es un sinónimo de *tipo*.
- Es un conjunto de mensajes que se le puede enviar.
- Es estática
  - Debe tener una única responsabilidad (Buscamos clases lo más cohesivas posibles, sin ir al otro extremo)

En rojo lo que me falta por copiar

En amarillo lo que ya copie

## ¿Qué es un objeto?

Un **objeto** puede tener

- **Datos internos** (lo que le proporciona el estado, un estado son el valor de las variables en un momento dado),
- **Métodos** (para proporcionar un comportamiento) e
- **Identidad** es decir cada objeto puede ser diferenciado de forma unívoca de cualquier otro objeto (cada objeto tiene una dirección de memoria exclusiva.)

Es dinámica.

Todo lo real puede ser modelado por un objeto, **todo es un objeto**.

Un objeto es una instancia de una clase.

## ¿Qué es un algoritmo?

Un **algoritmo** es una secuencia de pasos ordenados para resolver un problema.

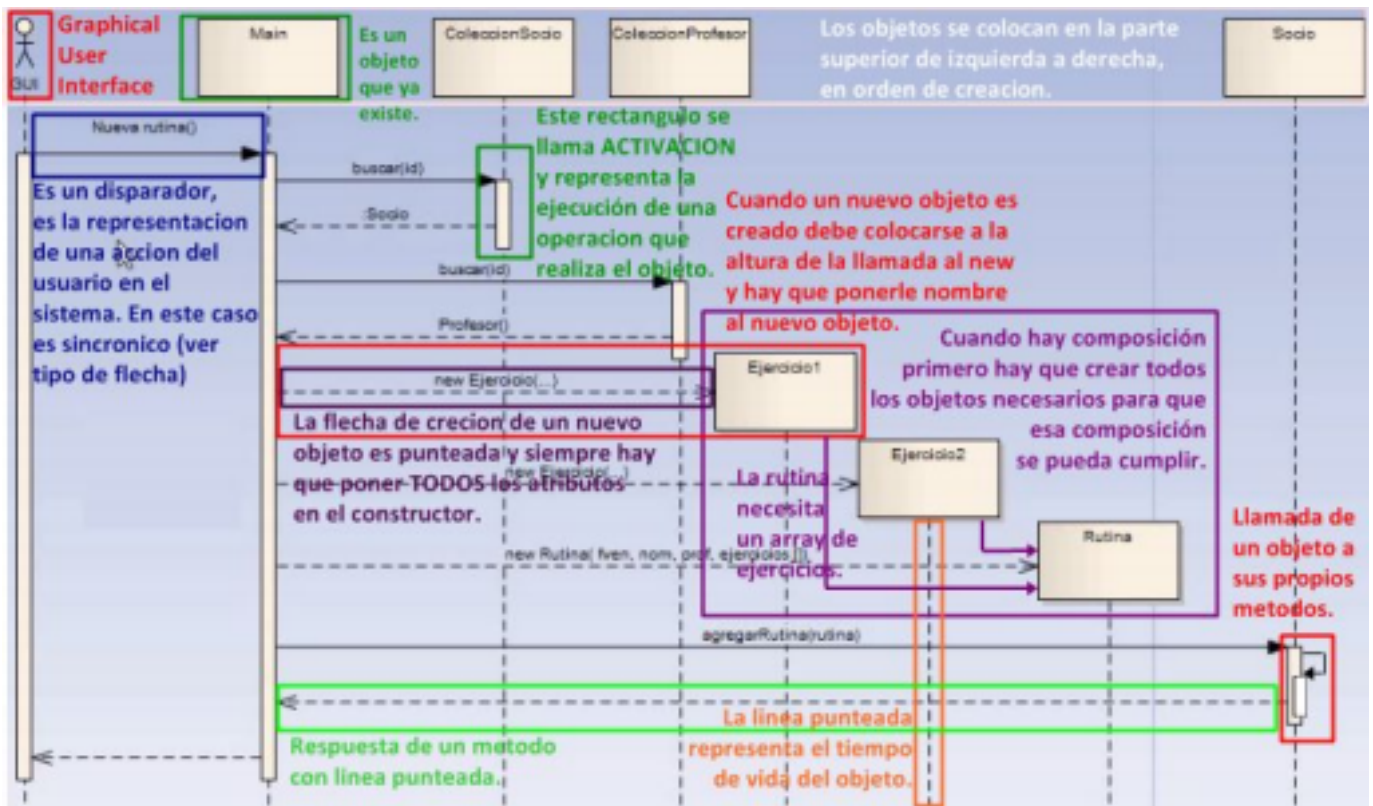
## ¿Qué es el polimorfismo?

Es la capacidad que tienen los objetos de una clase en ofrecer respuesta distinta e independiente en función de los parámetros (diferentes implementaciones) utilizados durante su invocación

## 1. Definición de diagrama de secuencia

Un **diagrama de secuencia**

- Modela la comunicación entre objetos en memoria.
- Muestra la forma en la que interactúan los objetos y el orden de ejecución a lo largo del tiempo.
- Ordena a nivel tiempo las peticiones entre objetos.



a. Diferencia entre mensaje sincrónico y asincrónico

**Mensaje sincrónico:** el objeto esperará la respuesta del mensaje antes de continuar.

**Mensaje asincrónico:** el objeto no esperará la respuesta antes de continuar.

i. De ejemplos

**Mensaje sincrónico:** getters y setters

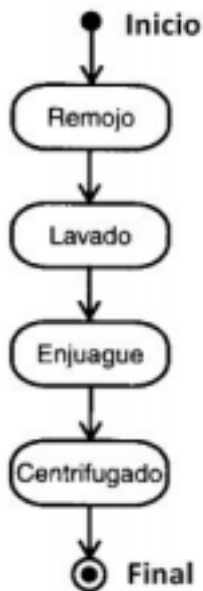
**Mensaje asincrónico:** enviar un mail

## 2. Definición de diagrama de estados

Un **diagrama de estado**

- Muestra los estados en los que puede encontrarse un objeto junto con las transiciones entre los mismos.
- Modela los cambios de estados de un único objeto en el orden que se ejecutan.
- Vamos a tener tantos diagramas de estados como objetos tenga nuestro sistema.

a. De un ejemplo  
**Lavarropas**



El círculo relleno simboliza el punto inicial.

Los cuadros con borde redondeado representan los estados.

El círculo con el borde sin rellenar representa el punto final.

Un **evento** es un acontecimiento importante por ejemplo en el ejemplo del lavarropas inicio/final y en el teléfono descolgar/colgar.

Un **estado** es una condición de un objeto en un momento determinado.

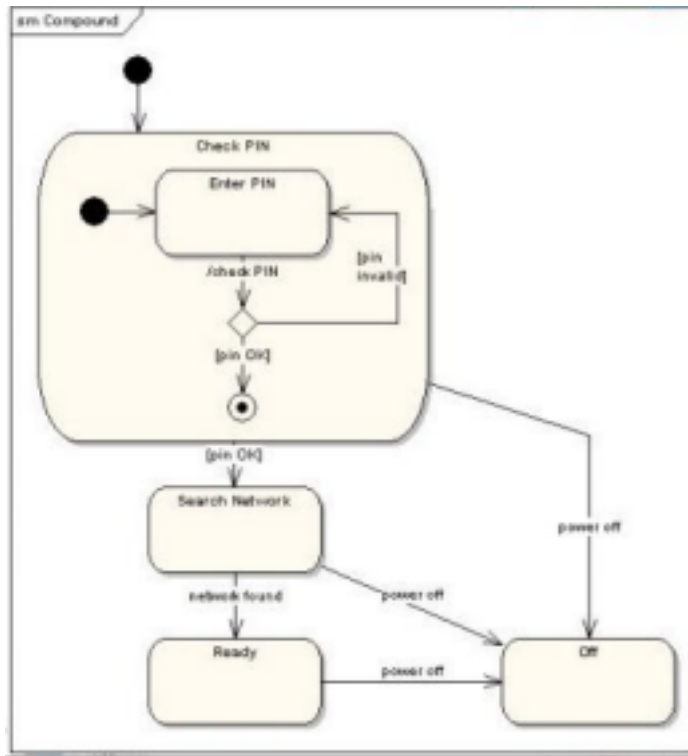
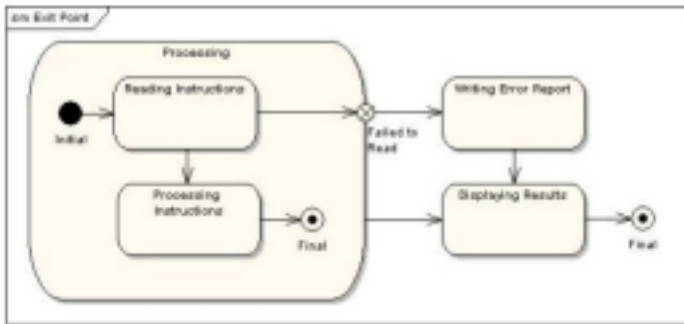
Una **transición** es una relación entre dos estados. Un evento puede ser una transición



b. ¿Qué es un estado compuesto?

Los **estados compuestos** contienen un diagrama de estado dentro de él. Esto se usa cuando un estado está compuesto por más de un estado o si hay algo que se quiera resaltar.

i. De un ejemplo

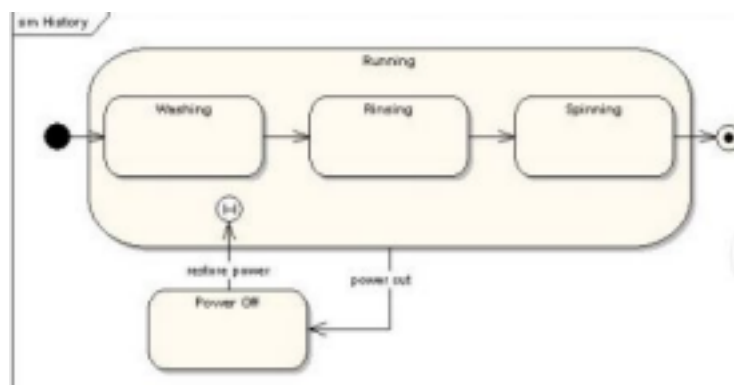


c. ¿Qué es un estado histórico?

Un **estado histórico** permite recuperar el estado anterior de cierto conjunto de estados. Por lo tanto, se utiliza para estados compuestos.

i. De un ejemplo

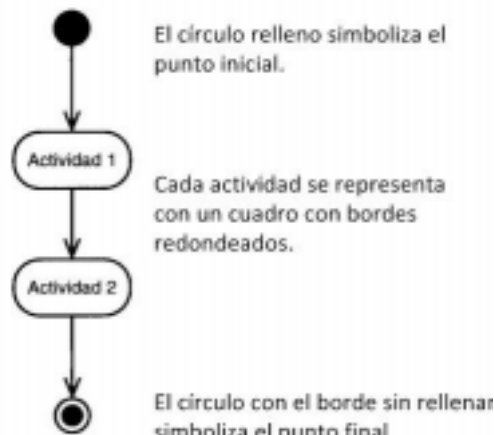
Si se corta la luz en medio de la ejecución proceso de lavado de un lavarropas, cuando se vuelve a prender, este revisa su estado histórico y reinicia desde ese estado. Esto quiere decir que usa este estado histórico como backup.



### 3. Definición de diagrama de actividades

#### El **diagrama de actividades**

- Muestra una versión simplificada de lo que ocurre durante una operación o un proceso.
  - Da idea de funcionalidad
  - Es una representación de alto nivel del funcionamiento de un sistema o caso de uso. •
- Es una extensión de un diagrama de estados.



## Vero Uhrich Final Programación 2

a. De un ejemplo.

## EJEMPLO - CREAR E IMPRIMIR UN DOCUMENTO

1. Abrir la aplicación para el procesamiento de textos.
2. Crear un archivo.
3. Guardar el archivo con un nombre en una carpeta.
4. Teclear el documento.
5. Si se necesitan ilustraciones, se abre la aplicación relacionada, se generan los gráficos y se colocan en el documento.
6. Si se necesita una hoja de cálculo, se abre la aplicación relacionada, se crea la hoja y se coloca en el documento.
7. Se guarda el archivo.
8. Se imprime el documento.
9. Se sale de la aplicación de oficina.

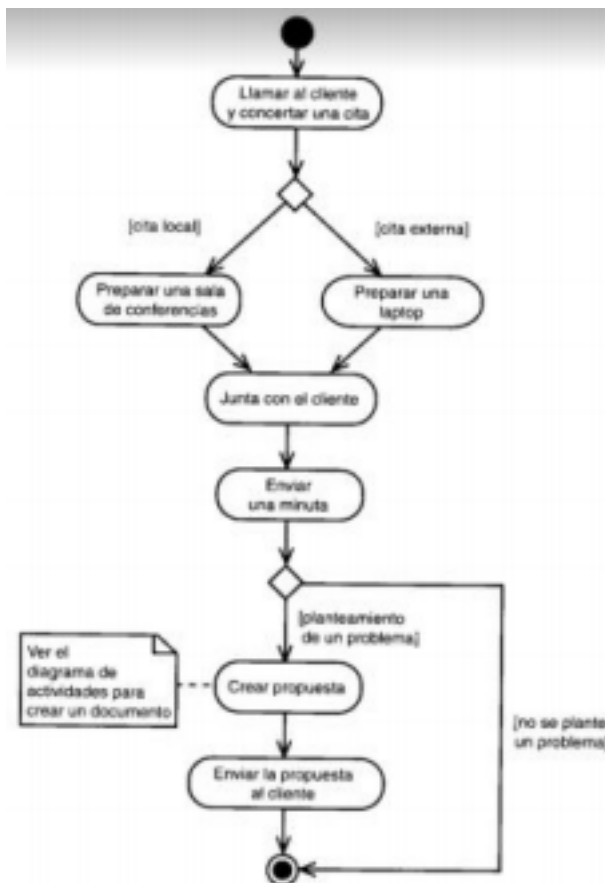


5

Caso de una firma de consultoría y el proceso de negociación involucrado en una junta con un cliente.

1. Un vendedor hace una llamada a un cliente y concreta una cita
2. Si la cita es en la oficina del consultor, los técnicos prepararán una sala de conferencias para hacer la presentación.
3. Si es en la oficina del cliente, un consultor preparará la presentación en una laptop.
4. El consultor y el vendedor se reunirán con el cliente en el sitio y a la hora convenidos.
5. El vendedor crea una minuta.
6. Si la reunión ha planteado la solución de un problema, el consultor creará una propuesta y la enviará al cliente.

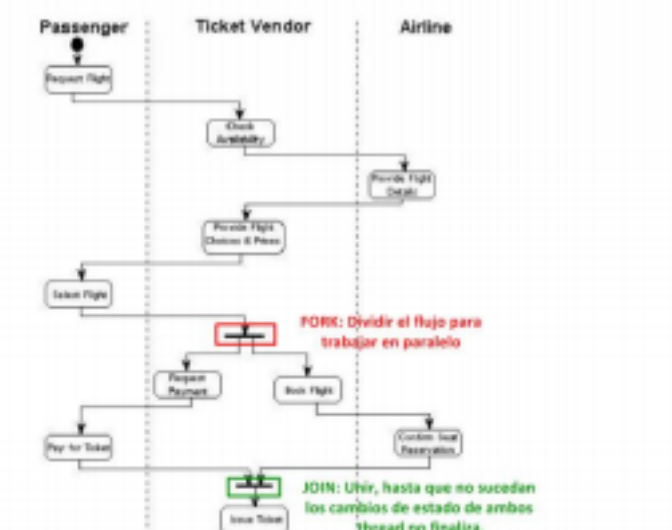
Vero Uhrich Final Programación 2



b. ¿Qué es el marco de responsabilidad?

- El **marco de responsabilidad** es otra forma de modelar un diagrama de actividades. Permite:
- Separar el diagrama en segmentos paralelos y observar claramente las responsabilidades de cada rol.
  - Ver perfiles de seguridad y transiciones entre actores.

## EJEMPLO - COMPRA DE VUELO







6

Vero Urrich Final Programación 2 4. Diferencia entre clase abstracta e interfaz.

DIFERENCIAS	
INTERFAZ	CLASE ABSTRACTA
Modela un comportamiento	Puede o no tener comportamiento
Define un conjunto de mensajes sin comportamiento	Puede o no tener métodos abstractos
Permite herencia múltiple	NO permite la herencia multiple
NO tiene estado interno (no tiene atributos)	Puede o no tener estado interno
No tiene constructor	No tiene constructor (no se puede instanciar)
Todos los métodos son públicos para que los puedan instanciar desde la clase que lo implemente.	Sigue el standard de visibilidad de las clases

a. ¿Para qué usaría cada una?

INTERFAZ:

- Cuando no se necesita lógica interna.
- Cuando necesito modelar un comportamiento y que quien lo implemente tenga la responsabilidad de concretar los métodos.
- Cuando necesito herencia múltiple.
- Cuando no necesito guardar estado interno.

CLASE ABSTRACTA:

- Si necesito solo modelar y no concretar métodos, osea solo tener métodos abstractos y no concretos.
- Si necesito métodos abstractos y concretos en una clase.
- Si tenemos un conjunto de reglas de negocio entonces es razonable tener una clase abstracta que lo maneje, este conjunto de reglas de negocio es difícil de separar para convertir la clase abstracta en interfaz.
- Si tenemos atributos que no podemos bajar a las clases por estas reglas de negocio, entonces necesitamos una clase abstracta


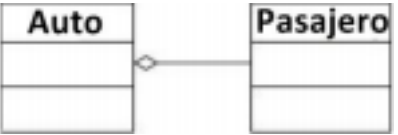



b. De un ejemplo de cada una **TODO**

5. Diferencia entre Asociación, Agregación y Composición

Es el tipo de asociación más débil.	Es un tipo de asociación intermedio.	Es el tipo de asociación más fuerte.
No hay dueño, cada objeto vive independiente.	Tiene <b>dueño temporal</b> , si el dueño desaparece este puede seguir viviendo..	Necesita de objeto débiles para vivir, su tiempo de vida está determinado por el objeto que lo incluye.
Un objeto puede estar asociado a más de un objeto.	Un objeto puede estar asociado a más de un objeto.	Tiene reglas de negocio obligatorias.

a. De un ejemplo de cada una.

6. Explique la relación de Dependencia (o Delegación)

Es **dependencia** cuando una clase usa objetos de otra clase, pero no se los guarda.

Es **delegación** cuando una clase devuelve objetos de otra clase. La delegación sirve para modularizar otros objetos.

a. De un ejemplo



**Generalizacion:** Responde siempre a la sentencia “es un” (herencia)

8



**Vero Uhrich Final Programación 2**

**Realización:** Es una clase que implementa una interfaz.



## 7. Definición de cohesión

La **cohesión** es el grado de relación que tiene el objeto con sus responsabilidades. Solo deben haber métodos funcionales en una clase, no deben haber métodos que no estén siendo usados. **Buscamos tener alto grado de cohesión.** Se aumenta la cohesión haciendo que el objeto implemente exclusivamente las responsabilidades que le corresponden. Además, de esta manera será más independiente del resto.

a. ¿Cuáles son los 2 niveles más altos?

**Cohesión funcional:** Los métodos de la clase están agrupados porque todos contribuyen a una responsabilidad bien definida.

**Cohesión perfecta:** Es **atómica**, la clase solamente tiene un único método (significa que no puede reducirse más).

b. De un ejemplo de cada uno.





## 8. Definición de acoplamiento

El **acoplamiento** es el nivel de dependencia entre las clases.

El objetivo perseguido es que dentro del sistema cada clase sea muy independiente y se comuniquen con el resto de las clases mediante una interfaz pequeña (con pocos parámetros) y bien definida, con **el menor acoplamiento posible**. Más parámetros, más nivel de acoplamiento.

Para tener un **bajo nivel de acoplamiento**:

- No hay que pasar clases hijas, sino las de más alta jerarquía.
- No hay que usar parámetros excursionistas.
- No hay que pasar clases como parámetro ya que hay que saber cómo está implementada.
- No hay que dejar atributos públicos.
- No hay que usar flags (esto da detalles de implementación por lo que rompe el encapsulamiento).

a. ¿Cuáles son los niveles?

**De marca o por estampado:**

- Ocurre cuando una de las clases de la aplicación es pasado como parámetro en un método. • La recomendación es no pasar formas de datos compuestas (clases) cuando el objeto solamente necesita uno o dos campos. Esto produce que se ensanche la interfaz, e incremente el costo de mantenimiento.
- Otro de los riesgos que esto produce es que un objeto pueda manipular y cambiar valores que no utiliza.

**De contenido:**

- Ocurre cuando un objeto modifica datos internos de otro objeto, rompiendo el principio de encapsulamiento.
- Esto se soluciona poniendo privados todos los atributos de las clases, y que él mismo modifique su estado mediante los getter y setters.

**De control:**

- Ocurre cuando un objeto le pasa a otros datos con la intención de controlar su lógica interna. (uso de flags).
- Cuando esto ocurre significa que un objeto conoce los detalles de implementación del otro objeto en forma implícita y se pierde independencia.
- Un buen diseño debe dejar que los objetos decidan qué hacer en base a su estado.

**De subclases:**

- Ocurre cuando una clase cliente tiene una referencia a la subclase en lugar de tener la referencia a la superclase. Esto aumenta la dependencia.

**De datos:**

- Ocurre cuando dos o más objetos intercambian información mediante el uso de argumentos simples (primitivos).
- Cuando más argumentos tiene un método más alto es el nivel de acoplamiento. Se debe reducir al mínimo posible el acoplamiento entre dos objetos no pasando más argumentos de los que verdaderamente utiliza, los estrictamente necesarios.

- Se recomienda evitar el uso de datos “excursionistas”, haciendo referencia a conjuntos de información que pasan de objeto en objeto sin ser utilizados hasta que alcanzan su destino.

b. De un ejemplo de cada uno. **TODO**

Una colección es una especie de tabla de datos en memoria. Necesitamos que sea único para asegurar que el repositorio sea uno solo, no haya duplicados e integridad en la información.

a. ¿Cuál es su relación con el Patrón Singleton?

Las Colecciones siempre son Singleton.

## Unidad 2

### 10. Definición de Arquitectura de Software

La **Arquitectura de Software** es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución. ○

¿Qué es la Arquitectura de Software?

La **Arquitectura de Software** es como debemos pensar un sistema a nivel macro. no solo el lenguaje de programación que voy a usar sino también la base de datos, el tipo de almacenamiento, la tecnología, como va a evolucionar esa tecnología. todo lo que hace que un sistema funcione.

Alude a la vista estructural general del software de alto nivel y el modo en que la estructura ofrece una integridad conceptual al sistema. (Entender el concepto general del sistema)

- Define estilo o combinación de estilos (arquitectónicos) para una solución.
- Se concentra en requerimientos no funcionales, todo lo que no está relacionado a la lógica del negocio pero es indispensable para que funcione. (Seguridad, escalabilidad, UX, que soporte cierta cantidad de peticiones). Esto es esencial para éxito o fracaso de un proyecto ○ ¿Qué no es la Arquitectura de Software?

- No es igual en la academia y en la industria
  - La academia propone modelos que la industria puede probar, aceptar o refutar. La Academia propone modelos y la industria dice cuáles sirven y cuáles no.
  - La industria implementa soluciones, que representan casos de estudio que luego son analizados y compartidos como casos de éxito o de mala práctica. Y esto varía todo el tiempo, la tecnología evoluciona.
- Está erróneamente relacionada con modelado Orientado a Objetos. Podemos tener una arquitectura que no esté orientada a objetos, híbridos, funcionales, así que no solo es orientado a objetos.
- Diseñada o documentada únicamente con UML. Yo puedo diseñar la arquitectura en lo que quiera. ● Erróneamente vinculada exclusivamente a la metodología. La arquitectura está totalmente desligada a la metodología que utilice (Agile por ejemplo)

### 11. ¿Para qué son los estilos arquitectónicos?

- Los **estilos arquitectónicos** definen ciertos patrones o conjunto de reglas que permiten aplicarse en las aplicaciones.
- Esto permite tener una lista de estilos y evaluar todas las alternativas que tengo con sus ventajas y desventajas conocidas ante diferentes conjuntos de requerimientos no funcionales. Por ejemplo, si necesito 24 mil peticiones no voy a utilizar una base de datos relacional.
- Los estilos arquitectónicos sirven para sintetizar estructuras de soluciones, ya están probados y no tenemos que reinventar los estilos sino utilizar los que ya existen.

### 12. Definición de arquitectura en capas

Las **arquitectura en capas** es un conjunto de niveles o capas con responsabilidades, en la cual cada nivel más bajo está más cerca al lenguaje de máquina.

○ ¿Qué es un sistema en capas puro?

En los **sistemas en capas puros**, cada capa solo puede comunicarse con la vecina. Aunque esta solución aunque puede ser menos eficiente en algunos casos facilita la portabilidad de los diseños.



## Vero Uhrich Final Programación 2

- ¿Cuáles son las 3 capas?  
Las capas separan responsabilidades en el sistema.  
Las capas son:



- De un ejemplo



### 13. Definición de arquitectura MVC

MVC consiste en tres tipos de objetos:

- El modelo es el objeto de aplicación,
- La vista (la capa de presentación) es su representación en pantalla y

- El controlador es el modo en que la la vista reacciona a la entrada del usuario.

Antes del MVC, los diseños de interfaces de usuario tendían a agrupar estos objetos en uno solo, no hay cohesión y mucho acoplamiento. Con MVC, se separa en objetos para incrementar flexibilidad y la reutilización.

MVC desacopla las vistas de los modelos estableciendo entre ellos un protocolo de comunicación de suscripción/notificación (gracias al uso del patrón Observer).

Una vista debe asegurarse de que su apariencia refleja el estado del modelo (todos los datos de mi lógica de negocio). Las vistas deben poder actualizarse y mostrar estos cambios cada vez que cambia mi lógica de negocio.

- Cada vez que cambian los datos del modelo, este se encarga de avisar a las vistas que dependen de él.

- Como respuesta a dicha notificación, cada vista tiene la oportunidad de actualizarse a sí misma. ●
- Así es posible asignar varias vistas a un mismo modelo para ofrecer diferentes presentaciones.

- ¿Cuáles son los 3 componentes?

Hoy MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario.

Se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.







- Definición de MVC pasivo y activo.

**MVC pasivo:** aquel que no notifica los cambios en el modelo, por lo que en el diagrama no iría la flecha entre la Vista y el Modelo. **No iría la flecha verde.**

(No tiene la posibilidad de recibir actualizaciones automáticas)

**MVC activo:** el modelo notifica a la vista sobre los cambios; por lo que en ese caso sí irían las flechas entre ellos. Acá es donde cobra importancia el Observer y el MVC en sí, el modelo está habilitado para avisarle a la vista que hay cambios. **Aca si está la flecha verde.**

- De un ejemplo: LOGIN **TODO**

#### 14. Modelo de 3 capas vs MVC

MODELO DE 3 CAPAS	MODELO MVC
El modelo de capas <b>es más general</b> , no define qué componentes debo utilizar.	MVC <b>obliga a utilizar sus componentes</b> , que suelen ser implementados como clases.
Puede usarse con lenguajes que no sean orientados a objetos. La capa no habla de un lenguaje en particular.	Se utiliza con lenguajes orientados a objetos.
Los eventos del usuario son recibidos directamente por la Capa de Presentación.	En MVC hay una capa anterior, una clase que carga la vista. Los eventos del usuario son recibidos primero por el controlador, nunca con una vista. <b>VENTAJA</b>
La capa de datos está acoplada a la base de datos. Si elijo SQL, todo lo que programo está en SQL. Si se decide cambiar a Oracle, tengo que cambiar todo el código.	El modelo está desacoplado a la base de datos. Aca los datos están separados del modelo. El modelo está en una capa más alta e implementa una interfaz común para el acceso a los datos independiente al lenguaje que haya elegido para los datos. Por lo que si quiero cambiar casi no hay

	costo, solo cambia el código de un método. <b>VENTAJA</b>
--	--------------------------------------------------------------

La capa de presentación se comunica directamente con la capa lógica. Es poco escalonado, la capa de lógica es inmediatamente la que le sigue.	Las vistas no se comunican directamente con el modelo. La vista se comunica con el controlador, la lógica está más atrás. Separa más las responsabilidades. <b>VENTAJA</b>
Capa de presentación sin estandarización de vistas.	Las vistas pueden estar compuestas por otras vistas. Puedo tener una vista que contenga la vista de Header, Footer y todas las que quiera, <b>VENTAJA</b>
	Una aplicación basada en MVC tiene n capas, donde tendremos las capas de la vista, el controlador, y el modelo siendo una agrupación básica de componentes, aunque la aplicación podría tener más capas asociadas a otras funciones o servicios. (Puedo tener capas dentro de una capa). <b>VENTAJA</b>
	Soporta la suscripción a eventos gracias a la versión activa, que implementa el patrón Observer. <b>VENTAJA</b>

Usar modelo de 3 capas solo para aplicaciones de escritorio, usar MVC en caso de ser posible, salvo que sea un proyecto heredado.

#### 15. Definición de Patrón de Diseño

Un **patrón de diseño** describe un problema que ocurre repetidas veces en nuestro entorno y da una solución genérica a ese problema no trivial.

- ¿Cuáles son sus 4 elementos esenciales?

Los 4 elementos esenciales son nombre, problema, solución y consecuencias.

- i. Explique cada uno

##### **Nombre:**

Describe en una o dos palabras un problema de diseño junto con sus soluciones y consecuencias. Por convención se usan los nombres originales en inglés.

Ejemplos:

- Observer (Observador)
- Strategy (Estrategia)
- Abstract Factory (Fábrica Abstracta)

##### **Problema:**

- Describe cuándo aplicar el Patrón.
- Explica el problema y su contexto.
- A veces el problema incluye una serie de condiciones que deben darse para que tenga sentido aplicar el Patrón.

##### **Solución:**

- Describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones.
- La solución no describe un diseño en concreto, sino que un Patrón es como una plantilla que puede aplicarse en muchas situaciones diferentes.

##### **Consecuencias:**

- Son los resultados, las ventajas e inconvenientes de aplicar el Patrón.

- Son fundamentales para evaluar las alternativas de diseño y comprender los costos y beneficios de aplicar el Patrón.
- Las consecuencias de un Patrón incluyen su impacto sobre la flexibilidad, extensibilidad y portabilidad de un sistema.
- Criterios de clasificación

La **clasificación de los Patrones** sigue dos criterios:

- **Propósito:** Refleja que hace un Patrón. Para que fue concebido.
- **De creación:** Tienen que ver con el proceso de creación de los objetos o sea como hacemos el new de una forma mucho más desacoplada y como mucha más cohesión y menos acoplamiento. Hacer el new en el main está mal. Ejemplo: Factory Method y Singleton.
- **Estructural:** Tiene que ver con la composición/asociación/agregación de clases u objetos, como un cambio estructural me empieza a resolver ciertos problemas. Ejemplo: Decorator y Facade.
- **De comportamiento:** Como las clases y objetos interactúan entre si y como se reparten la responsabilidades. Ejemplo: Observer, State y Strategy. En estos cambia el estado del objeto.
- **Ámbito:** Especifica si el Patrón se aplica principalmente a clases u objetos. Si se aplica a clases es estático y se aplica a objetos es dinámico.
- **Clase:** Relaciones entre las clases y sus subclases. Estas relaciones se establecen a través de herencia, de modo que son relaciones estáticas (fijadas en tiempo de compilación). Tener en cuenta que cuando se declara una clase que hereda de otra clase, esta **relación es estática** ya que una vez que se ejecuta el programa, esa clase no puede ser modificada. **La herencia es estática** porque es fijada antes de ejecutar el programa. Una **relación dinámica** es una relación que se puede cambiar en tiempo de ejecución. Los objetos son relaciones dinámicas y las clases estáticas porque al objeto lo declaro en tiempo de ejecución, entonces los objetos son dinámicos porque viven en memoria desde el momento que empezó la ejecución, puedo tocar su estado interno. Al contrario, cuando empiezo a ejecutar el programa no puedo cambiar un nombre o hacer que herede de otra clase, no puedo cambiar nada de una clase. Ejemplo: Factory Method
- **Objeto:** Tratan las relaciones entre objetos, que son dinámicas ya que pueden cambiarse en tiempo de ejecución. Ejemplo: Singleton, Decorator, Facade, Observer, State y Strategy.



Permite asignar responsabilidades adicionales a un objeto dinámicamente, haciendo de cuenta que cambia la piel del objeto. Esto proporciona una alternativa a la herencia simple para extender funcionalidad. Puedo sacar, agregar, juntar cosas en tiempo de ejecución.

En vez de usar la herencia simple usaremos agregación para componer los objetos complejos. A veces queremos añadir responsabilidades a objetos individuales en vez de a toda una clase. Este patrón permite customizar objeto por objeto. Una forma de añadir responsabilidades es a través de herencia. Por ejemplo, heredar un borde de otra clase pondría un borde alrededor de todas las instancias de la subclase. Sin embargo, esto es inflexible, ya que la elección del borde se hace estáticamente. Un cliente no puede controlar cómo y cuándo decorar el componente con un borde. No todos los objetos van a ser iguales por lo que debo permitir que cambien en cualquier momento, si estar ligado a un molde que es fijo y estático.

El objeto **decorador** encierra el componente en otro objeto que añada el borde. Esto es transparente para sus clientes porque el decorador se ajusta a la interfaz del componente que decora. El decorador maquilla al objeto, sin cambiar su funcionalidad, solo cambia la experiencia externa.

(Sabor, precio)

El decorador mediante la agregación o composición va a reenvía las peticiones al componente y puede realizar acciones adicionales (tales como dibujar un borde) antes o después del reenvío.

Esto permite anidar decoradores recursivamente y permite un número ilimitado de responsabilidades añadidas.

## Estructura



Este patrón se puede aplicar:

- Para añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos. • Para responsabilidades que pueden ser retiradas.
- Cuando la extensión mediante herencia no es viable, me obliga a saber que tipo de objetos voy a declarar. Este patrón viene a resolver este problema, no sabemos qué tipo de objeto vamos a declarar porque se

resuelve en tiempo de ejecución.





## Diagrama de secuencia

```
ComponenteVisual ventanaDecorada = new DecoradorBordeNegro(new DecoradorScrollHorizontal(new VentanaSimple()));  
ventanaDecorada.dibujar();
```



Otros ejemplos son:

- Un negocio que vende pizza: A la pizza base le agrego condimentos
- Starbuck: Al cafe le agrego cosas

Solo se le cambia la vista al objeto, la lógica es cambiado por el State

## Ventajas

1. **Más flexibilidad que la herencia estática:** se pueden añadir y eliminar responsabilidades en tiempo de ejecución gracias a la composición de objetos.
2. **Evita clases cargadas de funciones en la parte de arriba de la jerarquía:** “pagamos solo por lo que se necesita”. En vez de tratar de permitir todas las funcionalidades inimaginables en una clase compleja y adaptable, podemos definir primero una clase simple y luego añadir funcionalidad incrementalmente con objetos Decorador. **La funcionalidad puede obtenerse componiendo partes simples.**

## Desventajas

1. **Muchos objetos pequeños:** un diseño que usa el patrón Decorator suele dar como resultado sistemas formados por muchos objetos pequeños muy parecidos. Aunque dichos sistemas son fáciles de adaptar por parte de quienes los comprenden bien, pueden ser difíciles de aprender y de depurar.

Es importante que la clase Componente se mantenga ligera, debería centrarse en definir una interfaz, no en guardar datos. Por lo que usualmente debe declararse como una interfaz. La definición de cómo se representan los datos debería delegarse en las subclases.

Podemos pensar en el Decorator como un patrón que cambia la piel del objeto, es decir, solo cambia la parte exterior del objeto, NO CAMBIA LÓGICA.

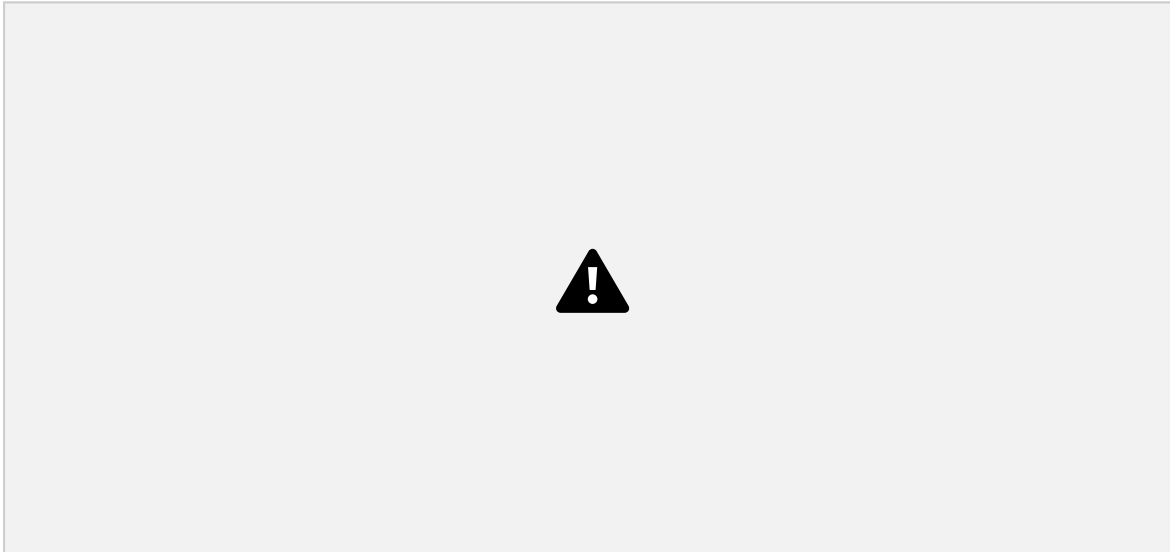
# PATRON OBSERVER

Ámbito de objeto y de comportamiento



El propósito del Patrón observer es definir una dependencia de uno a muchos entre objetos, de forma que cuando el objeto cambie de estado (se modifica al menos un atributo) se notifique y se actualicen automáticamente todos los objetos que dependen de él. En todos los casos donde haya un subscriptor y algo que informa se puede aplicar un Patrón Observer. En toda aplicación que necesita mostrar datos en tiempo real consumida de una única fuente pero que tiene que ser mostrada en varias Vistas, se puede aplicar un Observer.

Cuando se divide un sistema en una colección de clases que empiezan a cooperar, es necesario mantener una consistencia entre los objetos relacionados. Aca van a haber clases que definen los datos de las aplicaciones y otras que definen la representación de la información que deben poder reutilizarse de forma independiente.



Las vistas no conocen al sujeto pero se comportan como si lo hicieran. Cuando el usuario cambia la información en el formulario, el gráfico de barra muestra los cambios, al igual que la tabla y el gráfico de torta. El sujeto es nuestra fuente de datos, lo observable, en este caso la hoja de cálculo. Cuando se modifica una celda del excel, el sujeto le va a enviar un mensaje a cada uno de los observadores que se suscribieron al él. El sujeto tiene una lista de los observadores y cada observador puede hablar con el sujeto para pedirle algún tipo de información. El observador le puede pedir lo que cambio, ya que se le notificó que hubo un cambio pero no se le indicó cual fue el cambio.

## Estructura

### SUJETO (Observable):

- Tiene agregación con observadores por lo que puede tener muchos observadores suscritos o ninguno. • Es la entidad a la que queremos observar y enterarnos cuando tiene cambios. Necesitamos un comportamiento común para que todos los sujetos tenga la misma interfaz, los mismo métodos. • **Es clase abstracta** sin lógica de negocio.
- Tiene métodos genérico que me permiten suscribir, desuscribir y notificar (método que dispara la notificación)
- Cuando llamamos al notificar() se dispara la notificación. Este recorre el array de observadores y llama al actualizar() de cada uno. Tener en cuenta que solo notifica que hubo un cambio pero no le dice cual es el cambio. Por lo que el actualizar() debe pedirle al sujeto concreto el estado en caso de necesitarlo. **SUJETO**

### CONCRETO:

- **Es una clase concreta** que va a implementar los métodos que no estén implementados, si hay métodos que ya están implementados no va a hacer falta.

### OBSERVADOR (Observer):

- **Es una interfaz** común para todos los que se quieran actualizarse en base a un sujeto que tiene solamente el método actualizar().
- Obliga a que todos los que están interesados en obtener actualizaciones del sujeto implementen el método actualizar(). Si implemento esta interfaz tengo un array del tipo Observador, y puedo hacer polimórfica la llamada al notificar del Sujeto. Con un solo foreach puedo actualizarlos. Cada Observador concreto va a tener implementado su propio método actualizar() que sobrescribirá el método actualizar() del observador.

### OBSERVADOR CONCRETO:

- Pueden haber muchos y son las vistas.
- El método actualizar() se va a sobrescribir con lo que hace cada observador concreto.
- Se guarda una referencia al sujeto para poder pedirle los cambio. Esto lo va a hacer solo si se llama al método actualizar().



La suscripción de observadores es dinámica porque puede cambiar en tiempo de ejecución gracias a la agregación. La agregación es una relación dinámica.

El patrón permite 1 sujeto y 0 a N observadores.

Este patrón lo podemos aplicar:

- Cuando una abstracción tiene dos aspectos y uno depende del otro. Sin estar esperando un dato que lo tiene otro objeto, necesito encapsular estos aspectos en objetos separados para modificarlos y reutilizarlos en forma independiente.
- Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse. Si se actualiza algo necesito actualizarlo en un sistema.
- Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quiénes son dichos objetos. (Para evitar un fuerte acoplamiento)









Un cliente setea un estado, el sujeto concreto que hereda del sujeto abstracto (que no se dibuja) llama al `notificar()` que tiene un `foreach`, que llama a cada observador y por cada uno llama al `actualizar()` del observador concreto que es sobrescrito por la interfaz observador. El observador le pide al sujeto el estado y hace lo que tiene que hacer con esa información, por ejemplo mostrar los cambios de temperatura en una pantalla..

## Ventajas

1. **Modificar los sujetos y observadores de forma independiente.** Esto quiere decir que puedo añadir observadores sin modificar el sujeto u otros observadores. Es totalmente transparente por ser en tiempo de ejecución.
2. **Bajo acoplamiento entre Sujeto y Observador:** Al meter un nivel de abstracción más, todo lo que un sujeto sabe es que tiene una lista de observadores con la implementación de interfaz `update()`. Por lo tanto el sujeto no conoce la clase concreta de ningún observador, solo sabe que tiene que llamar al método `actualizar()`.
3. **Capacidad de comunicación mediante difusión:** cuando yo mando la notificación por un sujeto no necesita especificar su receptor, no hay que hacer ningún tipo de validación para enviar la notificación. El Observador es el que se encarga de suscribir. El sujeto está totalmente desacoplado de toda esa lógica, solo envía la notificación. La notificación se envía a todos los objetos interesados que se hayan suscrito a ella. Al sujeto no le importa cuántos objetos interesados haya, **su única responsabilidad es notificar a sus observadores.**

## Desventajas

1. **Sin historial de cambios:** Se usa un protocolo de actualización simple que no proporciona detalles acerca de qué se cambió en el sujeto. Se necesitarán protocolos adicionales para ayudar a los observadores a descubrir que cambió. Para la información completa, no muestra los cambios entre la info anterior y la actual. En ningún lado está guardada la comparación entre lo anterior y lo actual. Hay forma de obtenerlo, pero este patrón no lo contempla.
2. El modelo empieza a tener más clases (esto pasa con todos los patrones).

**Modelo push:** El sujeto envía a los observadores información detallada acerca del cambio, independientemente si se la solicitan o no. Cuando el sujeto actualiza le envía los cambios observador sin importar si al observador le importa o no. (Ejemplo de Hoja de cálculo)

**Modelo pull:** El sujeto no envía nada más que la notificación mínima, y los observadores piden después los detalles explícitamente. Cuando el sujeto actualiza no le envía los cambios al observador, envía una notificación de cambio y el observador debe pedir los cambios usando el atributo de estado que tiene guardado internamente. (Ejemplo: temperatura, no me sirve recibir medio grado de cambio de temperatura cada 30 segundos, por lo que voy a querer recibir el dato más paulatinamente. Haciendo alguna validación extra pido o no el cambio)

¿Cual usar?

Dependiendo el tipo de negocio me puede servir uno u otro.

## Unidad 3

19. Explique el Patrón Facade

### PATRON FACADE

La fachada agrega una nueva capa con los casos de usos más comunes implementando una interfaz unificada para acceder a ciertas funcionalidades de un subsistema (una porción del diagrama de clases). La fachada tiene como propósito proporcionar una interfaz unificada para este conjunto de interfaces de un subsistema. Esta interfaz de alto nivel (va a ser accedida desde el main) que hace que el subsistema sea más fácil de usar. La fachada tiene la misma lógica que el factory pero no del lado de la creación de objetos sino que sirve para abstraer la lógica del negocio.

Sirve para estructurar un sistema en subsistemas para reducir la complejidad al igual que el factory. Además tiene como objetivo minimizar la comunicación y dependencias entre subsistemas. Distintas fachadas se van a comunicar entre si. En vez de que los clientes se comuniquen con clases específicas, los clientes se van a hablar con fachadas que van a hablar con las clases específicas, las cuales se comunicarán con otras fachadas y las fachadas también se podrán comunicar entre ellas. Esto hace que se minimicen las dependencias.

Una forma de lograr esto es introducir un objeto Fachada, al igual que lo hace el Factory, que simplifique toda la interfaz para los servicios más generales del subsistema y proporcione una interfaz única. Antes teníamos muchos clientes que se comunicaban con el sistema con métodos de clases directamente. Por ejemplo, si quiero ver la cantidad de seguidores en facebook, debería leer todo el stack de Facebook para ver a qué método llamar... Las grandes aplicaciones no están programadas así, estas tienen fachadas. Las fachadas son una o varias clases que encapsulan los comportamientos más comunes del sistema y lo presentan a los clientes de forma amigable.



Esto quiere decir que cuando quiera ver los seguidores en facebook voy a tener un método obtenerSeguidores(urlPerfil) (una API) que me devuelva los seguidores. Así es como, al igual que el Factory, la Fachada se va a encargar de resolver todo lo que requiera hacer para obtener los seguidores. Además, el módulo de Facebook se deberá comunicarse con el módulo de seguridad, esto hace que las fachadas se comuniquen con



fachadas y no que la fachada tenga que comunicarse con el login y demás. La fachada se comunica con un método de autenticación de la Fachada de seguridad, este le devuelve lo que necesita y siga funcionando normalmente. La fachada además de ayudar al cliente, se ayudan entre ellas. Las fachadas tienen relación pero no conocen toda su implementación.

**Entonces la Fachada proporciona una interfaz única y simple, para que los servicios más generales están expuestos y sea más transparente, fácil y desacoplado para el cliente.**

Aplicamos Fachadas:

- **Cuando queremos proporcionar una interfaz simple para un sistema complejo.** Cuando los subsistemas van evolucionando y empiezan a tener más clases entonces se vuelven más complejos. Al aplicar patrones, es inevitable que crezca la cantidad de clases. Esto hace que el subsistema sea más reutilizable y fácil de personalizar, pero también lo hace difícil de usar para los clientes que no necesitan personalizarlo. Una fachada proporciona una vista simple del subsistema que resulta adecuada para la mayoría de los clientes. En las fachadas debemos capturar los casos de usos más comunes para que sean más fáciles de acceder por los clientes. Interactuar con un Factory es una capa más de abstracción por lo que casi acá probablemente haya una fachada.
- **Cuando hay muchas dependencias entre los clientes y las clases que implementan una abstracción.** Hacemos una fachada para desacoplar el subsistema de sus clientes y de otros subsistemas internos por lo que promovemos la independencia y la portabilidad entre subsistemas. Si hablo con una fachada en vez de hablar con todas las clases que están debajo de la fachada, desacoplo un montón de clases que antes dependían de una clase para que solo dependa de la fachada. Si yo cambio algo en las clases una fachada, por ejemplo en la de seguridad, cambio el login, voy a tener que cambiar el código de la fachada, pero solo en un lugar, no en todo mi sistema. El cliente no se entera de este cambio. El sistema se vuelve menos dependiente y ganamos menos acoplamiento y muchas más cohesión porque las fachadas tienen más responsabilidades y no todo el conjunto de clases que integra esta fachada.
- **Cuando queremos dividir en capas nuestros subsistemas.** Tendremos una capa de usuarios, o de negocio, o de datos, depende como lo queramos dividir. Se usa una fachada para definir un punto de entrada en cada nivel del subsistema. Si estos son dependientes, se pueden simplificar las dependencias entre ellos haciendo que se comuniquen entre sí únicamente a través de sus fachadas. Aca simplificamos la comunicación y evitamos las llamadas a clases particulares.

## Estructura

La **fachada** sabe qué clases del subsistema son las responsables ante una petición y delega las peticiones de los clientes en los objetos apropiados del subsistema.

Las **clases del subsistema** implementan la funcionalidad del subsistema y realizan las labores encomendadas por la fachada pero no debería agregar nuevas clases ni métodos porque el sistema ya lo hace, solo debemos agregar estos métodos a la Fachada.

La fachada conoce a las clases del subsistema pero las clases no conocen a la fachada, es decir, no tienen referencias a la fachada. La fachada es un nivel superior, solo los clientes la conocerán.





Dispositivo electrónico es una clase abstracta, ¿Podría ser interfaz?

Tiene un atributo, así que no podría, pero si saco la descripción y lo pongo en cada dispositivo electrónico, ¿Seguiría estando bien el diagrama? Sí, podría ser porque todos los métodos son abstractos y se van a definir en las clases concretas. Todos tienen encender() y apagar() y se implementan de forma diferente en las clases concretas. Entonces, podría ser una clase abstracta, pero si bajo la descripción a los distintos dispositivos electrónicos también podría ser una interfaz y permitir la herencia múltiple ya que el resto solo modela comportamiento que es implementado por las clases hijas. Si tuviera muchos atributos sería más pasar de clase abstracta a interfaz.

Conclusion: pensando en Java, lo pasaria a interfaz para así tener herencia múltiple. Por ejemplo Proyector sólo puede heredar de Dispositivo Electrónico, pero si fuera interfaz podría heredar de dos clases. Si tenemos un conjunto de reglas de negocio entonces es razonable tener una clase abstracta que lo maneje, este conjunto de reglas de negocio es difícil de separar para convertir la clase abstracta en interfaz.

## Diagrama de secuencias



# Ventajas

1. **Ocultar a los clientes los componentes del subsistema**, reduciendo así el número de objetos con los que tratan los clientes y haciendo que el subsistema sea más fácil de usar. Si exponemos los casos de uso más comunes el cliente está más contento que si tiene que leer todo el stack de métodos para ver como usar el sistema.

2. **Promueve un débil acoplamiento entre el subsistema y sus clientes**. Lo que permite modificar los componentes del subsistema sin que sus clientes se vean afectados. Si tengo un método en la fachada y este método cambia, el cliente no se entera del cambio (mientras no cambie la firma del método). Así desacoplamos el sistema y elevamos un nivel de abstracción.

3. Ayuda a estructurar en capas un sistema y las dependencias entre objetos y es mucho más ordenado y potable.

4. Si tengo un sistema muy grande puede ayudar a eliminar dependencias complejas o circulares. 5.

**No impide que las aplicaciones usen las clases del subsistema en caso de que sea necesario.**

De este modo se puede elegir entre facilidad de uso y generalidad.

○ ¿Los clientes solo pueden acceder a las clases a través de la fachada?

No, también pueden seguir accediendo a los métodos como lo venía haciendo. Aquellos clientes que necesitan acceder a los casos de uso menos comunes necesitarán ir más allá de la fachada y acceder directamente a los métodos de las clases. Aca hay una **diferencia con el Factory** ya que yo no estoy obligado a usar la fachada si no tengo el caso de uso que necesito. Siempre que se pueda usar la fachada es conveniente usarla.

20. ¿Cuál es la relación entre el Patrón Facade y el Singleton?

Normalmente sólo se necesita un objeto fachada. Por lo tanto, se suelen implementar como Singletons. La fachada suele ser Singleton porque se supone que el acceso a un sistema o subsistema desde métodos debe ser único, no quiero metodos repetidos pero dependerá de si la fachada tiene o no estado interno. Si una Fachada tiene estado interno o tiene estado interno pero puede ser utilizado por todo el sistema entonces puede ser Singleton. Si tiene guardado un estado interno, como por ejemplo datos de un usuario entonces no puede ser compartida por todo el sistema, por lo tanto no puede ser Singleton la Fachada.





21.

Explique el Patrón Factory Method

## FACTORY METHOD

- Es una fábrica de objetos. Implementa una interfaz para la creación de objeto pero usan el polimorfismo para dejar que sean las subclases quienes decidan qué hacer y qué clase instanciar. Vamos a tener una familia de fábricas pero cada familia concreta va a decidir qué clase instanciar.
- Permite que una clase delegue en sus subclases la creación de objetos. Una clase madre delega a las clases hijas la creación de objetos.

El objetivo de este patrón es desacoplar la lógica de creación de los objetos del cliente, vamos a dejar de tener new de objetos en el Main, un el método de creación va a contener el new.

Cuando empiezo a crear nuevos objetos, el factory se vuelve fundamental. Es importante pensar en siempre delegar responsabilidades, el main no debería tener la responsabilidad de crear objetos.

Podemos aplicarlo:

- Siempre que haya que crear objetos.
- Cuando no se no se puede saber con anterioridad que clase se debe crear. Cuando hay una familia enorme de objetos y no se el cliente cual de esas clases va a usar. Así delego la creación de objetos de manera seguro, transparente y fácil.
- Cuando queremos que las clases hijas sean la que creen las familias de objetos.

22. ¿Cuáles son las variantes de implementación?

## Variantes de implementación

La implementación de la clase Creador pueden ser de cuatro tipos:

### Métodos de fabricación parametrizados

Recibir por parámetro una variable que identifica el tipo de objeto a crear. Esto permite que los métodos de fabricación creen varios tipos de productos. El método de fabricación nuevoDocumento() recibe un parámetro, y por ende FabricarDocumento() también, y hay un condicional que pregunta qué tipo es y usa el constructor correcto para crear el objeto dependiendo de de ese flag. Esto genera un poquito de acoplamiento de datos, estoy acoplando el código a ese parámetro pero no se puede evitar esto. En este caso, la clase Creador puede ser clase abstracta o concreta.

Estructura





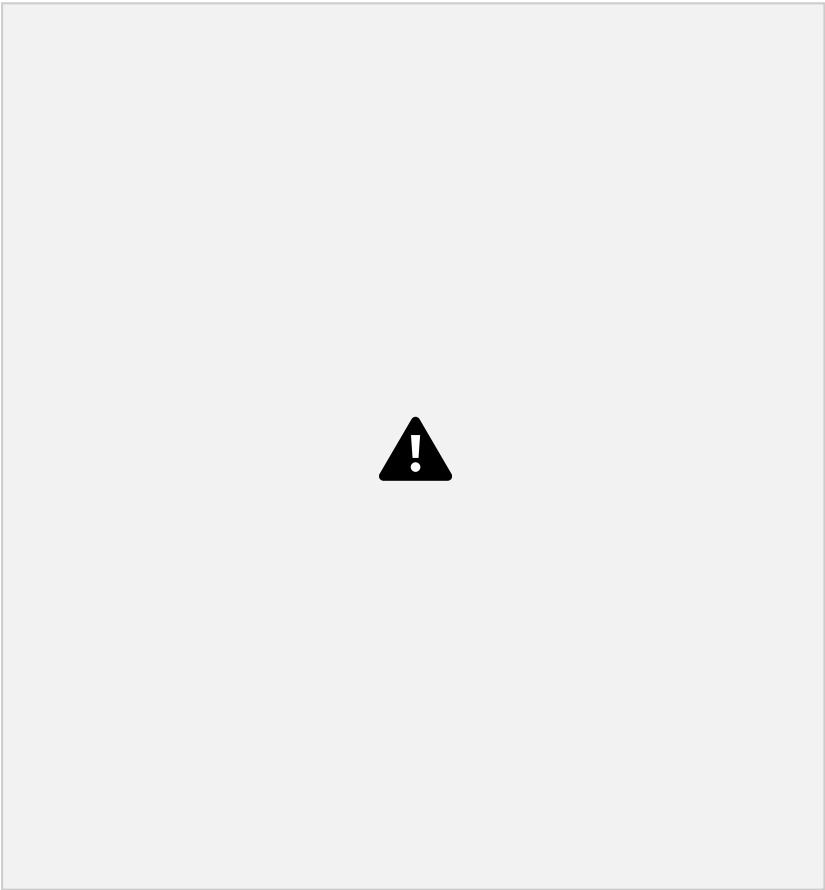
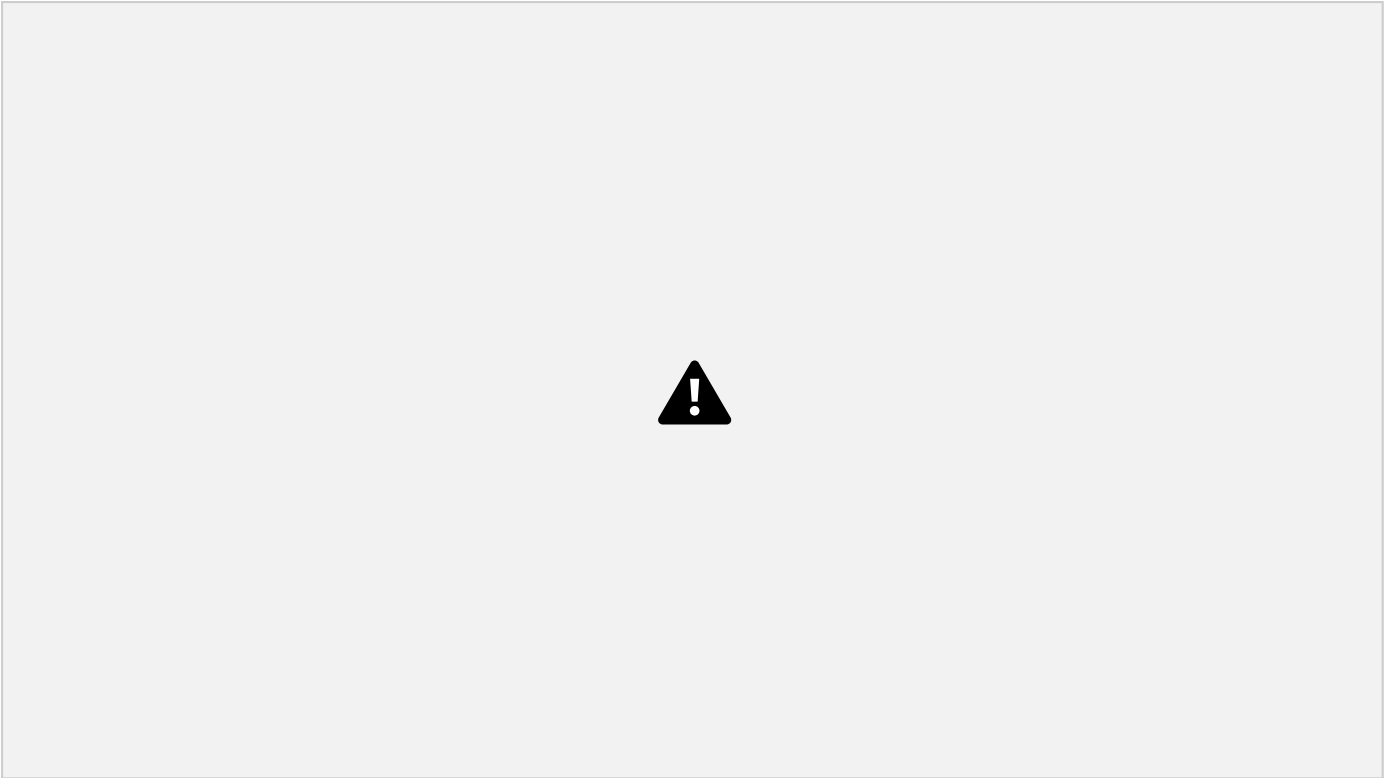
## **Abstracta**

El Creador es una clase abstracta y no proporciona una implementación para el método de fabricación que declara, el método ensamblador va a estar vacío. Con esto obligo a las fábricas concretas a implementar todos los métodos de fabricación, no hay fabricación por defecto.

## **Concreta**

Es lo contrario a la abstracta. El Creador es una clase concreta y proporciona una implementación predeterminada del método de fabricación. Por ejemplo, si el usuario no me dice el tipo de fabricación a usar entonces que por defecto haga un .doc, pero esto depende del negocio.







## Clase abstracta que define una implementación predeterminada

Es poco común. El Creador es una clase abstracta pero el método de fabricación es concreto y proporciona una implementación por defecto. Esta clase no se puede instanciar pero cuando se cree una clase CreadorConcreto puedo o no sobrescribir el método de fabricación, porque ya está escrito arriba. Entonces cuando se cree una fábrica ya hereda el comportamiento de la clase Creador. Si me olvide o por alguna regla de negocio ya tengo definido un comportamiento por defecto.

## Diagrama de secuencia



Los métodos de fabricación eliminan la necesidad de ligar clases específicas de la aplicación a nuestro código. En el main en vez de tener una clase específica, voy a llamar a un método para crear un objeto. Solo voy a tener una clase fábrica, si quiero agregar algo nuevo a la fábrica va a ser transparente para el cliente.

## Ventajas

1. **Flexibilidad:** Crear objetos dentro de una clase con un método de fabricación siempre es más flexible que hacerlo directamente porque encapsulo comportamiento, agrego una capa más de abstracción.
2. **Encapsulamiento:** Ocultan la lógica de creación, que en la realidad suele ser compleja.

## Desventajas

1. Se obliga al cliente a definir subclases de la clase Creador sólo para crear un producto concreto y esto puede no ser apropiado siempre. A veces puedo tener muchas fábricas para definir un solo producto.
2. Aumenta el número de clases.
3. En el tipo de implementación parametrizado hay un poquito de acoplamiento.

# PATRON SINGLETON

Tiene como propósito garantizar una única instancia de acceso a cualquier clase y proporcionar un único punto de acceso (global). Esto significa que si se quiere acceder a una clase en particular, esta sea única (por alguna regla de negocio) y tenga un interfaz común para que cualquiera que quiera acceder a esa clase use esa misma interfaz de manera global, para que no hayan distintas entradas ni distintas formas de acceder a lo mismo. Tengo una sola entrada controlada que me permite manejar esta única instancia de clase.

Singleton se usa cuando:

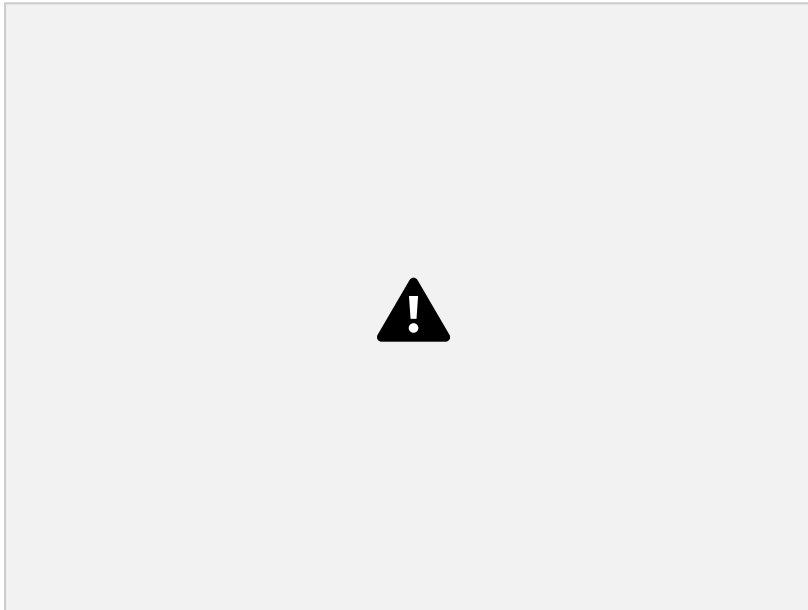
- Existe una regla de negocio que me diga que tengo que tener una única instancia de una clase.  
Ejemplo: Una cola de impresión, una conexión a una base de datos, una clase que graba en un log, cualquier tipo de clase que el negocio me dice que no se puede repetir.
- Esta clase será responsable de garantizar que haya una única instancia de ella misma. Esta clase se autoadministra y proporciona una interfaz para que cualquiera la pueda usar, dando a los clientes un punto de acceso conocido para obtener la instancia.
- También será responsable de que no se pueda crear cualquier ninguna otra instancia de esa clase que esa misma.

## Estructura





## Diagrama de secuencia



## Ventajas

**1. Acceso controlado a la única instancia:** Como la clase Singleton encapsula su única instancia, se tiene un control estricto sobre cómo y cuándo acceden a ella los clientes, ya que tengo métodos específicos. **2. Espacio de nombres reducido:** Es una mejora sobre las variables globales, ya que me olvido qué variables comparto y con qué módulos usando solo clases.

**3. Es fácil volver a permitir más instancias:** Si por algún motivo una clase donde se aplicó el patrón Singleton debe comenzar a permitir más de una instancia, no es complejo volver a permitir varias instancias, sólo debo modificar una clase sacando las cosas que hacen que sea única la Clase.

- ¿Es thread safe? ¿Por qué? ¿Cómo se puede solucionar?

## Desventajas

### No es thread safe.

Esto sucede porque si múltiples hilos quieren obtener una instancia, ambos podrían estar generando más de una, si acceden al mismo tiempo.

Para solucionarlo, se debe programar en Singleton de manera que su creación esté sincronizada y no ocurra más de una vez. Esto se hace poniéndole al método la etiqueta de sincronismo (en Java sincronizate) que genera una cola y se ejecuta uno detrás del otro, sin empezar otra ejecución sin que termine la anterior. De esta forma nunca van a entrar los dos juntos. Esto se puede usar cuando hay pocas peticiones ya que sino se crearía un cuello de botella. Entonces esto lo podemos utilizar cuando el número de cliente es un número razonable.

- ¿Qué significa lazy load?

Instancia usa **lazy load** (inicialización perezosa), lo que significa que la instancia no se crea y se almacena hasta que se accede a ella por primera vez. Esto quiere decir que la instancia siempre empieza en null para no desperdiciar memoria y solo hago el new cuando se hace la petición.

Otros ejemplos:

- Clases que permiten guardar Logs: Necesito que los logs se guarden en un único archivo. 40

# PATRON STATE

Así como el Decorator cambia la piel del objeto, el State cambia la clase, hace que el objeto sea una clase diferente cambiando su estado interno. Por lo tanto el State permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto pero en verdad lo que está pasando es que, gracias a la agregación, la herencia y el polimorfismo, cambia un objeto agregado por otro y pareciera que el objeto se comporta de manera diferente.

El Patrón State va a describir como un Contexto va a exhibir un comportamiento diferente dependiendo el estado en el que se encuentre.

State se usa:

- Cuando el comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado.
- Cuando tengo muchos ifs que dependen del estado del objeto. El Patrón State separa cada rama del if en una clase aparte permitiendo poder tratar al estado del objeto como un objeto que puede variar independientemente de otros objetos. Quitando acoplamiento y cohesión.

Los clientes pueden configurar un contexto con un Estado predeterminado. Esto hace que sus clientes hablen con el Contexto y ya no tengan que hablar con el Estado directamente.

Cualquiera de las subclases de Contexto o de EstadoConcreto pueden decidir cuál es el próximo estado y bajo qué circunstancias entonces...

**¿Cuales son las diferentes formas de implementar un State entre Contexto y Estado? ¿Quién define las transiciones entre estados?**

El Patrón State no especifica qué participante define los criterios para las transiciones entre estados.

## 1. El Contexto decide qué estado sigue a otro.

Ventajas:

Permite al Estado acceder al contexto si fuera necesario.

Desventaja:

Genera acoplamiento y responsabilidad del Contexto. Toda la lógica está en el Contexto.

Se usa cuando el contexto es simple, cuando al implementar las transiciones en los distintos estados se gana más dependencia y se pierde flexibilidad.

## 2. Cada Estado Concreto tiene la lógica de cual sigue a cual.

Generalmente es más flexible y conveniente que las subclases de Estado especifiquen su estado sucesor y cuándo llevar a cabo la transición (cuándo cambia el estado). **Esto requiere añadir métodos al Contexto para que los Estados puedan modificar el estado actual del Contexto.**

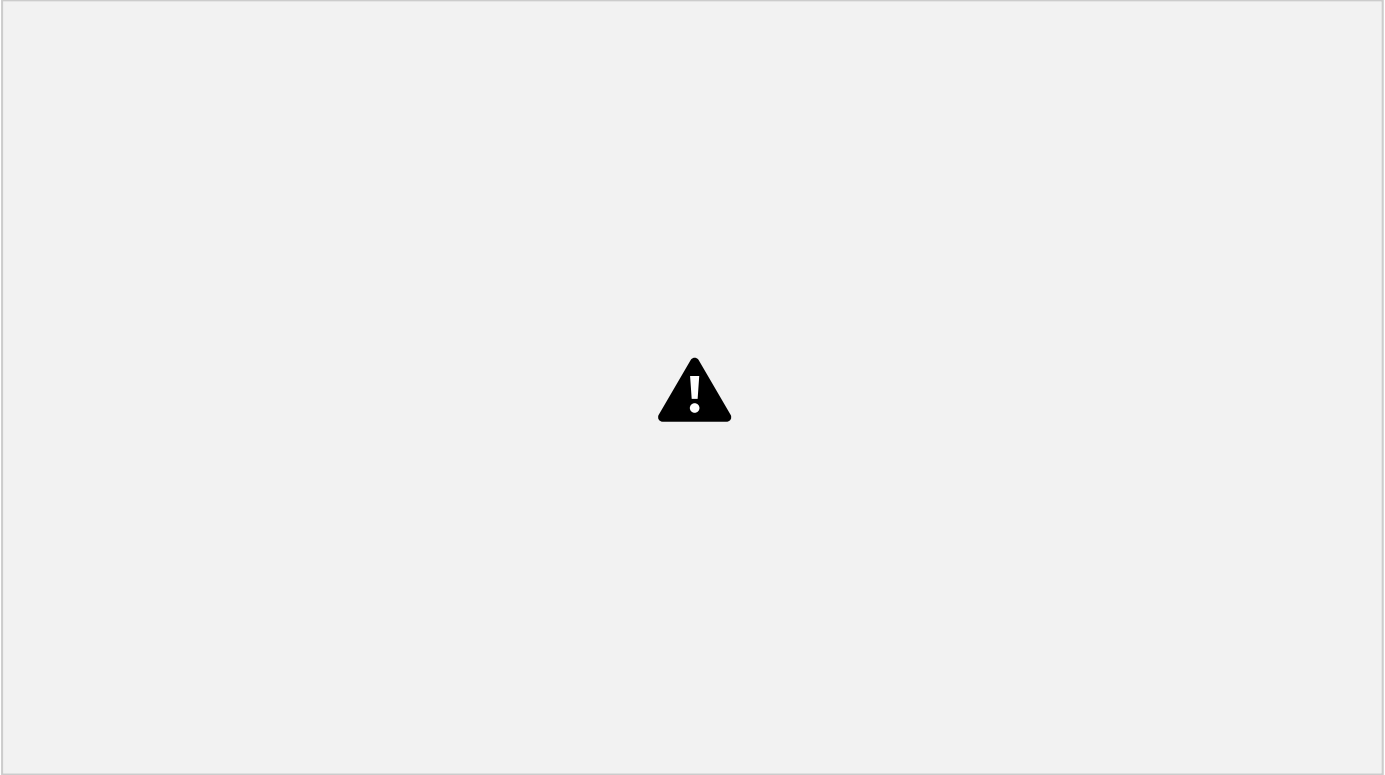
Ventaja:

Descentralizar la lógica de transición facilitando su modificación o extensión definiendo nuevas subclases de Estado.

Desventaja:

Una subclase de Estado va a tener dependencia con al menos otra subclase de Estado, ya que debe conocer su implementación. Es *al menos uno* porque de un estado se podría poder ir a dos estados.

Estructura













# Diagrama de secuencia



## Ventajas

1. **Localiza el comportamiento dependiente del estado y divide este comportamiento en diferentes estados:** en un objeto está todo el comportamiento asociado con un único estado. Ya que todo el código del estado se encuentra en una subclase EstadoConcreto. Esto permite añadir fácilmente nuevos estados y transiciones definiendo nuevas subclases.
2. **Desaparecen los if:** la lógica que determina las transiciones entre estados no reside en sentencias if sino que es polimórfica y está repartida en las subclases de Estado.
3. **Hace explícitas las transiciones entre estados:** es una forma de autodocumentar el código. Con mirar el diagrama de clases, ya se sabe cuales son las transiciones y estados posibles.

## Desventajas

1. **Incrementa el número de clases:** al distribuir el comportamiento para los diferentes estados en varias subclases de Estado, incrementa el número de clases y es menos compacto que una única clase.

El State es un Patrón comúnmente usado, ya que se puede aplicar en los escenarios que dependan de cierto comportamiento asociado a un grupo de estados.

Algunas aplicaciones son:

- Protocolos de conexión.
- Funciones de los editores gráficos (se definen estados para dibujar, seleccionar, renellar, etc) ○
- Funciones de los video players
- Estados de usuarios

25. ¿Cuál es la relación entre el Patrón State y el Singleton?

Los objetos Estado suelen implementarse como Singletons si no mantienen un estado interno. En el ejemplo de motivación, las subclases de EstadoTCP no mantienen un estado interno, de modo que pueden compartirse, por lo que solamente es necesaria una única instancia de cada una, por lo que dichos estados son Singletons.

# PATRON STRATEGY

Define una familia de algoritmos, encapsulando cada uno de estos algoritmos en clases y permite variar cada uno de estos algoritmos independientemente de los clientes que lo usan. Hay una nueva herencia y una nueva agregación o composición que permite cambiar dinámicamente un algoritmo.

- Si se codifican los algoritmos en las clases que se usan no es una buena práctica porque
- **Hay acoplamiento:** El código se vuelve más complejo si tienen que incluir dicho código, lo que los hace más grandes y más difíciles de mantener. Si tengo varios algoritmos para manejar la solución y los incluyo en el cliente entonces tengo más código, mas acoplamiento y es menos mantenible.
  - Los distintos algoritmos no van a ser siempre usados. **No tenemos por qué permitir múltiples algoritmos si no los vamos a usar todos.** Al igual que en el decorator no voy a usar todos los decoradores. Ac pasa lo mismo, no debemos incluir múltiples algoritmos sino lo vamos a usar a todos. Hay que pagar por lo justo y necesario
  - Al igual que en el decorator, si quiero agregar un nuevo decorator es fácil. Si los algoritmos están en el cliente es difícil agregar nuevos algoritmos o modificar existentes. El Strategy crea clases que encapsula estos algoritmos y hace que no estén en la clase Cliente.

Entonces un algoritmo así encapsulado se denomina una Estrategia.

El patron Strategy se usa:

- Cuando muchas clases que están relacionadas y solo difieren sólo en su comportamiento. Esto me indica que puedo separar en estrategias y permiten configurar una clase con un determinado comportamiento de entre muchos posibles. La idea es desacoplar ese comportamiento y ponerlo en una clase aparte.
- Cuando se necesitan distintas variantes de un algoritmo.
- Cuando un algoritmo usa datos que los clientes no deberían conocer. Con el Patrón Strategy evito exponer estructuras de datos complejas y dependientes del algoritmo.
- Cuando tengo una clase define muchos comportamientos con muchas sentencias condicionales en sus operaciones. Un buen código de programación orientado a objetos no debería tener condicionales. Cada if debería ser una Estrategia, así aprovechamos el polimorfismo, la agregación y la composición y decidimos en tiempo de ejecución que estrategia usar.

## Estructura









## Diagrama de secuencia



- ¿Cuáles son los enfoques para el acceso a los datos entre Contexto y Estrategia?
- Estrategia y Contexto interactúan para implementar el algoritmo elegido. Existen 2 alternativas: 50

1. **Un Contexto puede pasar a la estrategia todos los datos requeridos por el algoritmo.** Si tengo un Contexto para ordenar un array, el Contexto le pasa el array a la Estrategia y la Estrategia devuelve el array ordenado.
2. **El Contexto delega la responsabilidad a la Estrategia pasandose a sí mismo como parámetro de las operaciones de la Estrategia.** Esto permite a la Estrategia hacer llamadas al Contexto cuando sea necesario.

Entonces, un Contexto es una especie de Fachada, solo redirige peticiones de los clientes hacia la Estrategia. Los clientes normalmente crean un objeto EstrategiaConcreta, el cual pasan al Contexto. Por lo tanto **los Clientes interactúan sólo con el Contexto.**

Suele haber una familia de clases EstrategiaConcreta a elegir por el cliente.

El cliente, ¿Puede elegir las Estrategias a implementarse? Si, este patrón necesita del input del cliente. Es indispensable en este patrón que el cliente conozca qué hacen las estrategias, no su implementación.

i. Ventajas y desventajas.

## Ventajas

1. **Tengo familias de algoritmos relacionados:** Con la herencia saco factor común de la funcionalidad de estos algoritmos y lo encapsulo bajando el acoplamiento.
2. **Las estrategias eliminan las sentencias condicionales:** Cuando se juntan muchos comportamientos en una clase, es difícil no usar if para seleccionar el comportamiento correcto. Encapsular el comportamiento en clases Estrategia separadas elimina estas sentencias condicionales.
3. **Una elección de implementaciones:** Solo pago por lo que voy a usar, no instancio ninguna clase que no vaya a usar. Me da una elección flexible y liviana de que quiero implementar ya que las estrategias pueden proporcionar distintas implementaciones del mismo comportamiento y puedo usar solo una o muchas.
4. **Una alternativa a la herencia:** Encapsular el algoritmo en clases Estrategia separadas nos permite variar el algoritmo independientemente de su Contexto, haciéndolo más fácil de cambiar, comprender y extender. Esto se podría hacer con herencia pero es más difícil de comprender, mantener y extender. Además no se puede modificar el algoritmo dinámicamente. Vamos a tener muchas clases relacionadas cuya única diferencia es el algoritmo o comportamiento que utilizan, osea que el comportamiento queda ligado al Contexto.

## Desventajas

1. **Los clientes deben conocer las diferentes estrategias:** El cliente debe saber en que se diferencian las estrategias antes de seleccionar que va a utilizar.  
Los clientes pueden estar expuestos a cuestiones de la implementación por eso **sólo se debe usar Strategy cuando la variación del comportamiento es relevante a los clientes.**
2. **Costes de comunicación entre Estrategia y Contexto:** La interfaz de Estrategia es compartida por todas las clases de ella (EstrategiasConcretas), ya sea que el algoritmo que implementa sea trivial o complejo. Por lo tanto **es probable que algunos objetos EstrategiaConcreta no usen toda la información que reciben a través de dicha interfaz.** Esto significa que habrá veces que el Contexto crea e inicializa parámetros que nunca se usan, porque **el Cliente elige una estrategia que usa solo un parámetro y yo le mande los parámetros para todas las estrategias posibles.**
3. **Mayor número de objetos:** Las estrategias aumentan el número de objetos de una aplicación.

ii. ¿Qué enfoque es mejor?

Las necesidades del algoritmo concreto y sus requisitos de datos determinarán cuál es el mejor enfoque, no hay un mejor enfoque, depende la situación.

1) El Contexto puede pasar los datos como parámetros a las operaciones de Estrategia. **Ventaja**

Esto mantiene a Estrategia y Contexto levemente desacoplado, porque paso solo datos, no tengo dependencia a nivel método.

**Desventaja**

Contexto podría pasar datos a la Estrategia que esta no necesita. Este enfoque se usa cuando tengo parámetros de entradas similares en todas las estrategias, si voy a ordenar un array probablemente siempre le pase el array.

2) El contexto se pase a si mismo como parámetro y que la Estrategia pida los datos explícitamente al Contexto.

**Ventaja**

La Estrategia puede pedir exactamente lo que necesita.

**Desventaja**

El Contexto debe definir una interfaz más elaborada para sus datos, lo que **aumenta el acoplamiento entre Estrategia y Contexto**, ya que le estoy pasando a la Estrategia un objeto y tiene que conocer la implementación de los métodos del Contexto, y si estos cambian tengo que cambiar más código.

Esta Estrategia se usa cuando se tienen estrategias con parámetros de entrada muy distintos, así la Estrategia hable con el contexto y pida lo que tiene que va a usar.

## ¿Que pasa si hago opcionales los objetos estrategia?

Se puede simplificar la clase Contexto. Si no tiene sentido tener un objeto Estrategia, el Contexto comprueba si tiene un objeto Estrategia antes de acceder a él. Esto es parecido al Singleton, hace una especie de Lazy Load, y si existe lo usa, sino, el Contexto realiza el comportamiento predeterminado.

La **ventaja** es que los clientes no tienen que tratar con los objeto Estrategia a menos que no les sirva el comportamiento predeterminado. Por ejemplo al cargar la grilla de Mercado Libre por default los productos aparecen en orden de relevancia y si al cliente no le sirve lo cambia a ordenar por menor precio. De esta forma no fuerzo al cliente a elegir una estrategia. Además es Composición Contexto con Estrategia ya que necesita una estrategia para funcionar y esta estrategia es la definida por default. En otros casos la estrategia tiene que ser elegida si o si por el usuario y no tenemos una por default.

27. ¿Cuál es la relación entre el Patrón Strategy y el Singleton?

Las Estrategias Concretas pueden ser Singleton ya que se pueden instanciar solo una vez y ser utilizado por todo el sistema. Esto sucede solo si es Thread Safe, se genera una cola de ejecución, el método se ejecuta y no empieza otra ejecución hasta que no termina la ejecución anterior. Sino fuera Thread Safe se pisan las variables del algoritmo de ordenamiento.

