Patrones de diseño de comportamiento

Semana 9

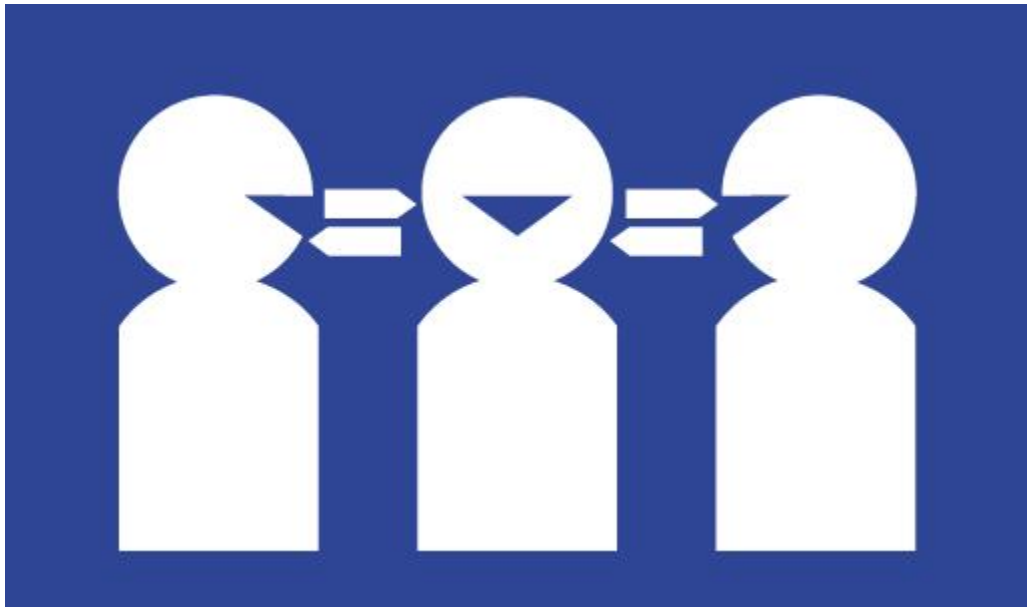
Agenda

- Patrones de comportamiento
- Observer
- Iterator
- Strategy
- Chain of Responsibility

Patrones de comportamiento

Patrones de Comportamiento

- Permiten asignar responsabilidades a objetos.
- Distribuyen comportamiento entre varias clases.
  - Mediante herencia
  - Mediante composición de clases.



Breve descripción de cada uno

- **Chain of Responsability**:

  Permite asignar una responsabilidad a una cadena de objetos

- **Command**:

  Encapsula una petición en un objeto.

- **Interpreter**:

  Permite especificar gramáticas (expresiones regulares).

- **Iterator**:

  Accede secuencialmente a objetos.

Breve Descripcion de cada uno

- **Mediator**:

  Coordina interacciones entre sus asociados.

- **Memento**:

  Captura el estado de un objeto y lo restaura posteriormente.

- **Observer**:

  Permite notificar cambios de estados a objetos dependientes

- **Strategy**:

  Encapsula un algoritmo dentro de una clase.

Observer

Observer

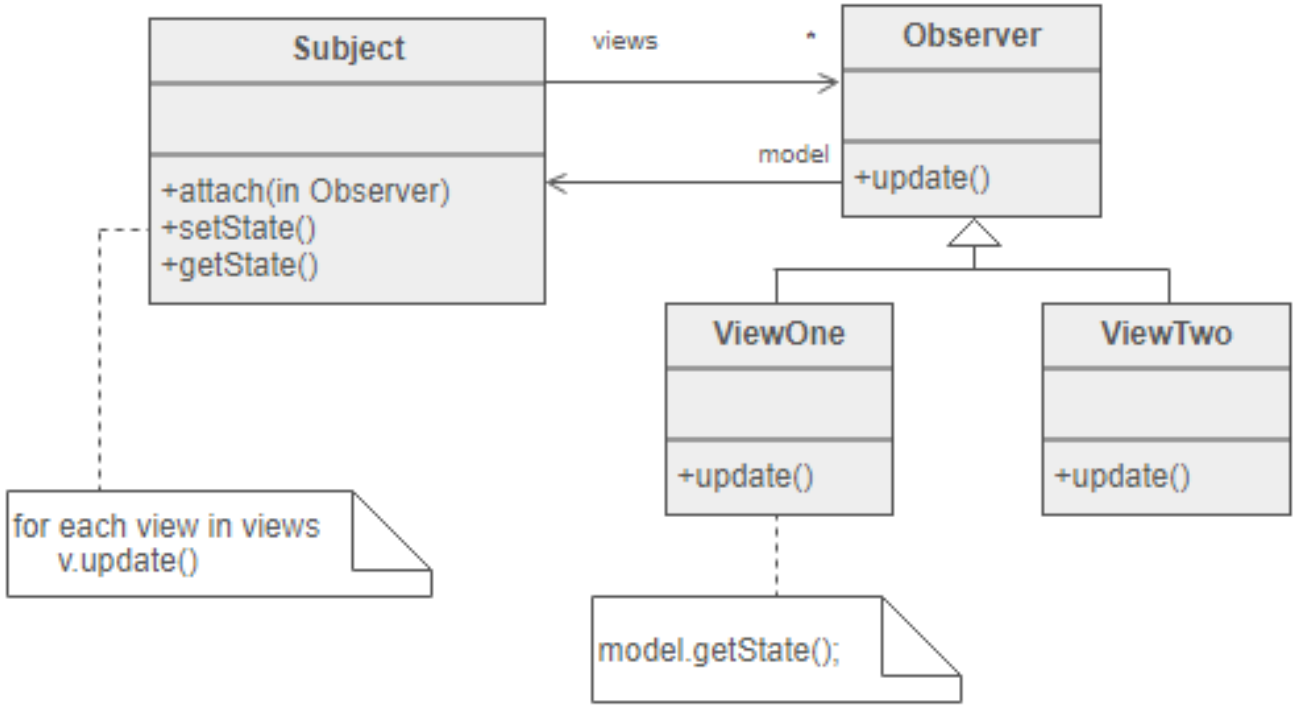
- Propósito
  - Define una dependencia uno a muchos entre objetos, de tal forma que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.
  - Es como la “Vista” en el modelo MVC
- Motivación
  - Ayudar a resolver problemas de escalabilidad de sistemas monolíticos.

Observer - Filosofía

- Un objeto, denominado *sujeto* (*Subject*) posee un estado. Cuando su estado cambia, es capaz de "avisar" a sus *subcriptores* (*Observers*) de este cambio de estado.

- De este modo, los objetos suscritos al objeto no tienen que preocuparse de cuándo se produce un cambio de estado: éste se encargará de informar de forma activa a todos aquellos objetos que hayan decidido suscribirse.

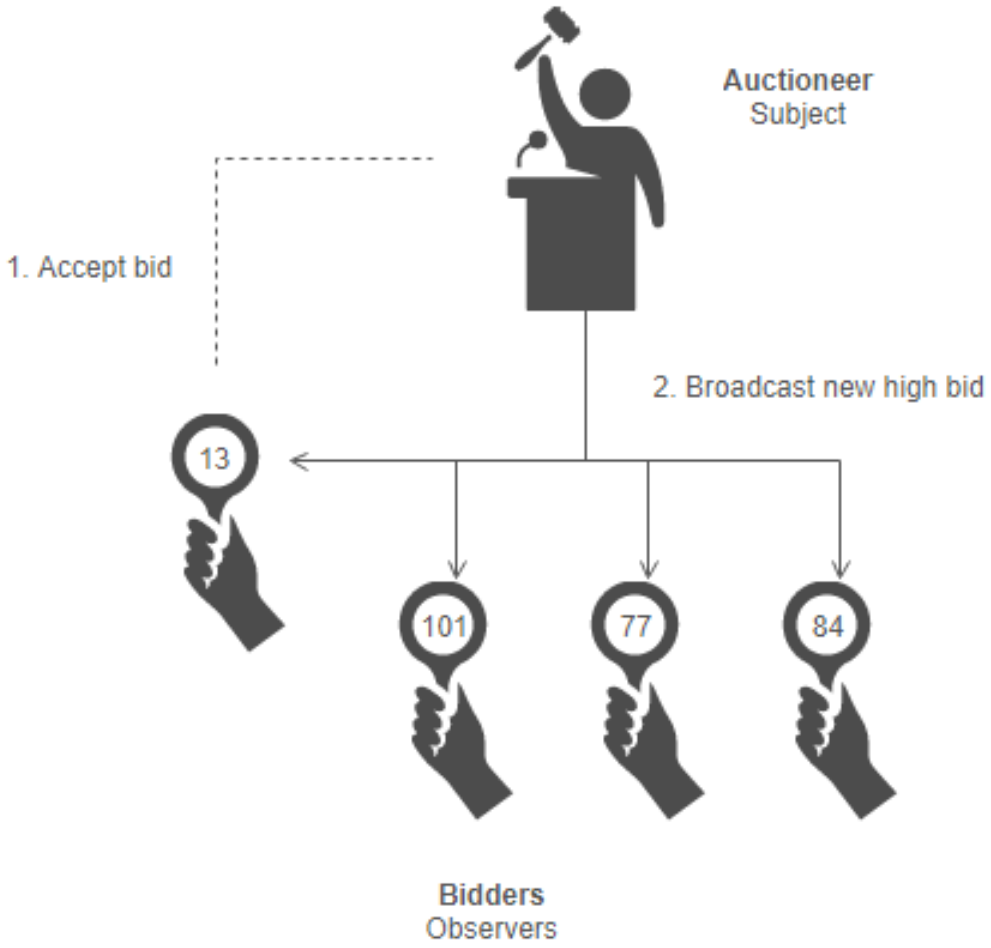
Observer - Implementación



Check list...

1. Diferenciar entre la funcionalidad core (independiente), de la dependiente.
2. Modelar la funcionalidad independiente como una abstracción ‘subject’.
3. Modelar la funcionalidad dependiente como una jerarquía ‘observador’
4. Asociar el Sujeto a la clase base del Observador.
5. El Cliente configura el número y tipo de observadores.
6. Los Observadores se registran (suscriben) con el sujeto.
7. El Sujeto transmite eventos a todos sus Observadores.
8. El Sujeto hace push de los cambios o el Observador hace pull.

Ejemplo


Auctioneer
Subject
1. Accept bid
2. Broadcast new high bid
13
101
77
84
Bidders
Observers

Observer - Ejemplos Reales

- Es el patrón base para creación de eventos en la implementación de interfaces de usuario.
  - Los *Event Handler* corresponden a los *Observers*
- Es la base del enlace de datos (data binding)
- Corresponde a las Vistas en el patrón MVC
- Ejemplo:
  - *https://howtodoinjava.com/design-patterns/behavioral/observer-design-pattern/*

Iterator

Iterator

- **Propósito**
  - Proporcionar una forma de acceder a los elementos de un objeto agregado de forma secuencial sin exponer sus detalles.

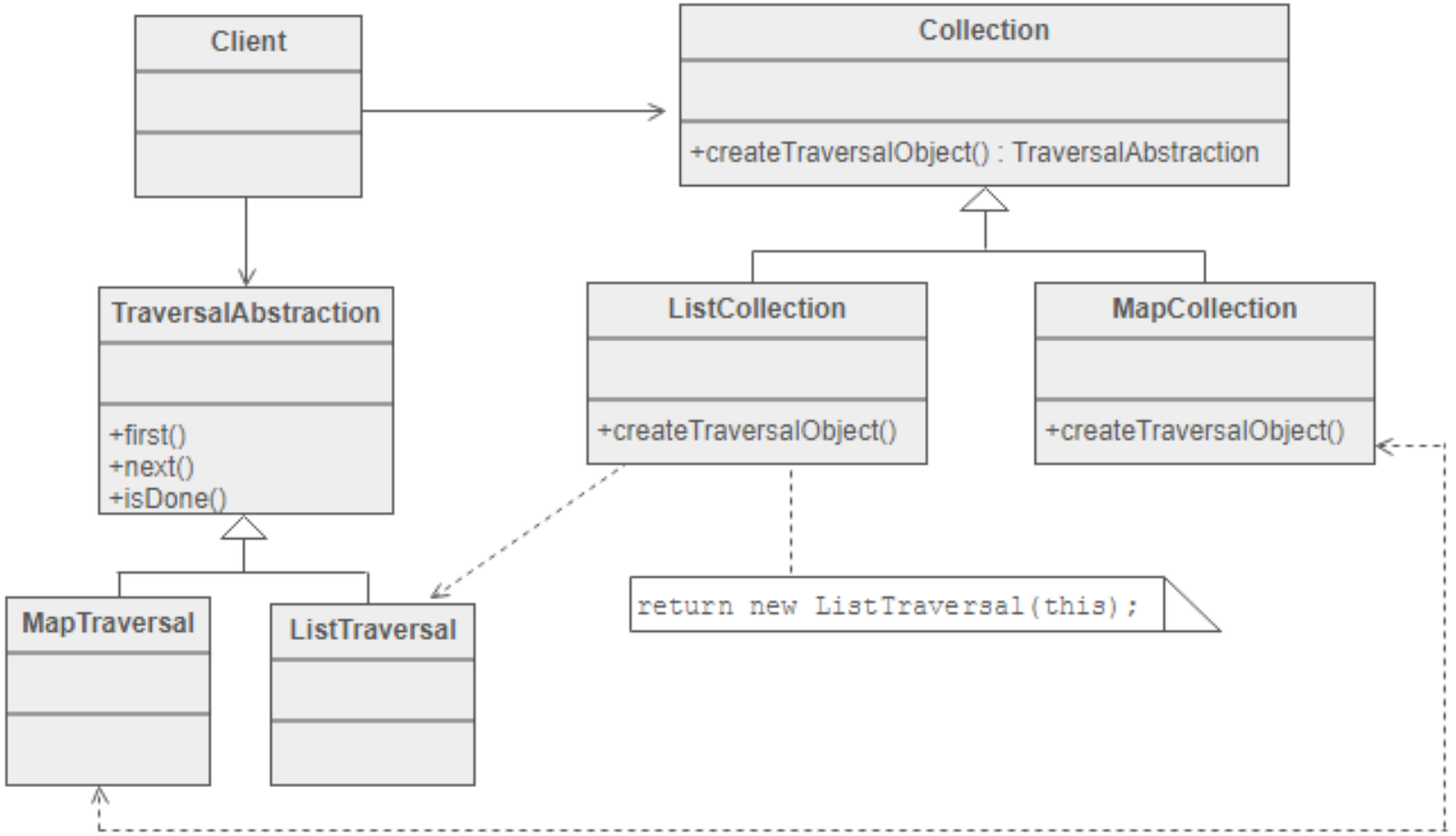
- **Motivación**
  - Abstraernos de detalles de implementación al momento de recorrer una colección de objetos.

Iterator – Filosofía

- Recorrer **secuencialmente una colección de objetos**, esto es, hacer uso de un proceso que sea capaz de **situarse en el primer elemento** de una colección y **obtener la información de ese contenido**.

- Es necesario **pasar del elemento actual al elemento siguiente**, obteniendo también su contenido. Es necesario implementar algún mecanismo que nos informe si hemos alcanzado el **final de la colección** para detener el proceso de iteración..

Iterator - Implementación



Check list…

1. Añadir el método create_iterator() a la clase de la colección.
2. Diseñar una clase “iterador” que encapsule los métodos para recorrer la colección.
3. Cliente pide a la colección crear el objeto iterador.
4. Cliente usa los métodos first(), isdone(), next() y current_item() para acceder a los elementos de la colección.
5. Ejemplo:
   - https://howtodoinjava.com/design-patterns/behavioral/iterator-design-pattern/

Strategy

Strategy

- **Propósito**
  - Definir una familia de algoritmos, encapsular cada una y hacerlos intercambiables.
  - Capturar la abstracción en una interfaz, colocar detalles de implementación en clases derivadas.

Strategy – Motivación

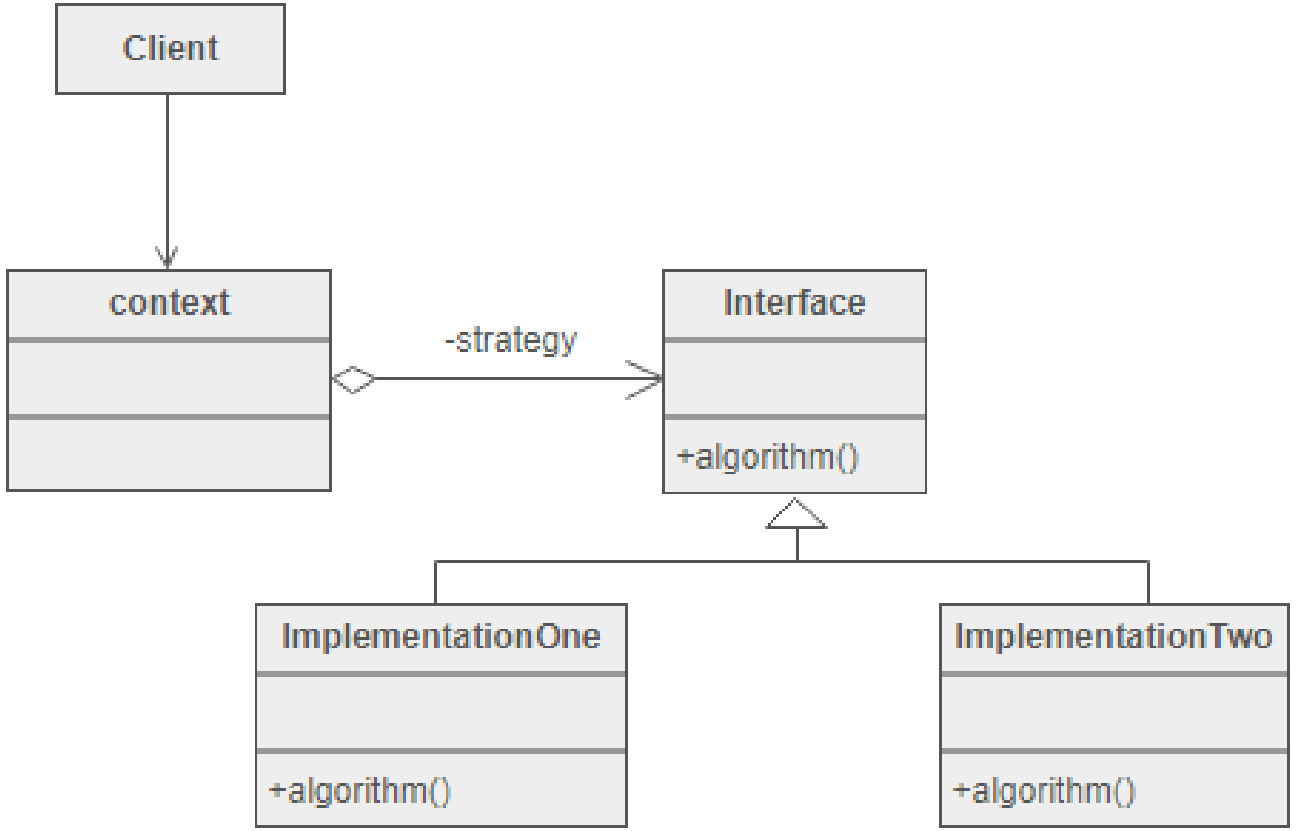
- Incluir el código de los algoritmos en los clientes hace que estos sean demasiado grandes y complicados de mantener y/o extender.
- El cliente no va a necesitar todos los algoritmos en todos los casos, de modo que no queremos que dicho cliente los almacene si no los va a usar.
- Si existiesen clientes distintos que usasen los mismos algoritmos, habría que duplicar el código, por tanto, esta situación no favorece la reutilización.

Strategy

- El nombre de este patrón evoca la posibilidad de realizar un cambio de estrategia en tiempo de ejecución sustituyendo un objeto que se encargará de implementarla.

- No nos preocupará el “cómo”. De hecho, ni siquiera nos importará “el qué”: la clase que actúa como interfaz del patrón únicamente tendrá que exponer el método o métodos que deberá invocar el cliente.

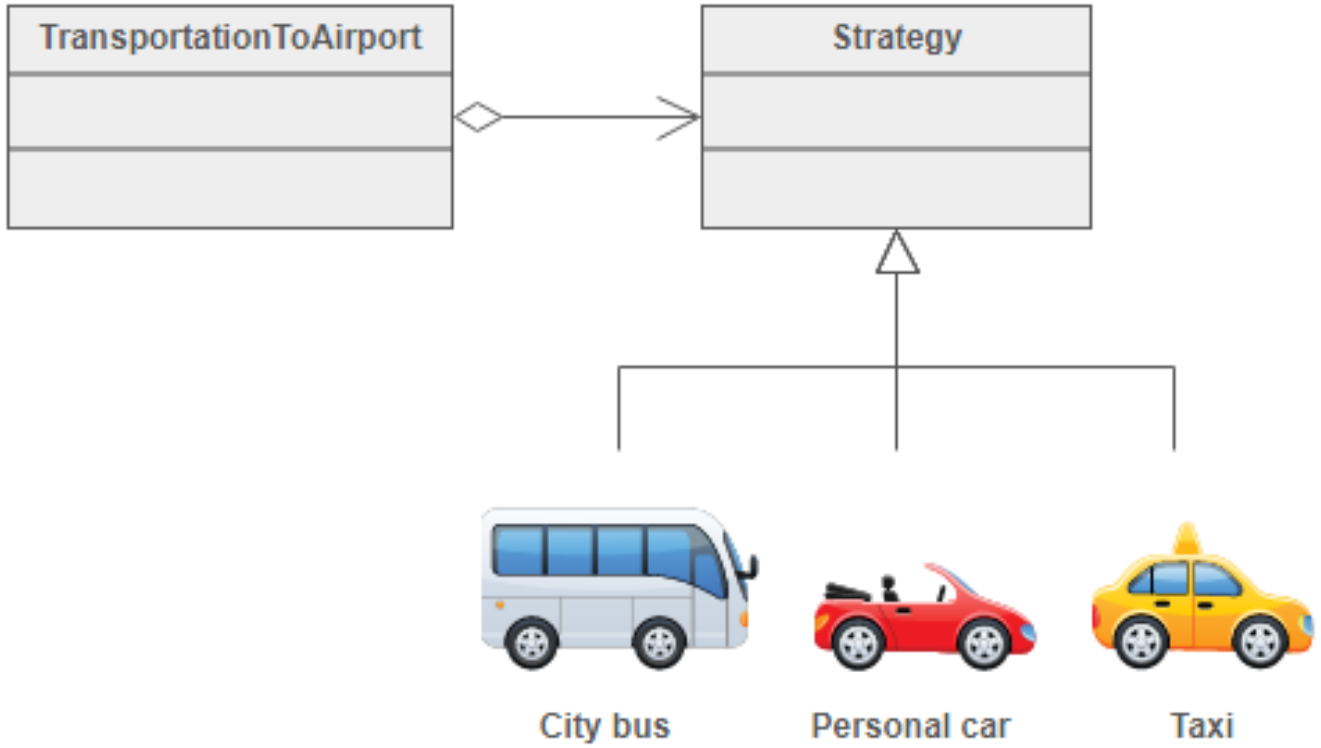
Strategy - Implementación



Strategy - Participates

- **Contexto** (*Context*) : Es el elemento que usa los algoritmos, por tanto, delega en la jerarquía de estrategias. Configura una estrategia concreta mediante una referencia a la estrategia necesaria.
- **Estrategia** (*Strategy*): Declara una interfaz común para todos los algoritmos soportados. Esta interfaz será usada por el contexto para invocar a la estrategia concreta.
- **EstrategiaConcreta** (*ConcreteStrategy*): Implementa el algoritmo utilizando la interfaz definida por la estrategia.

Example

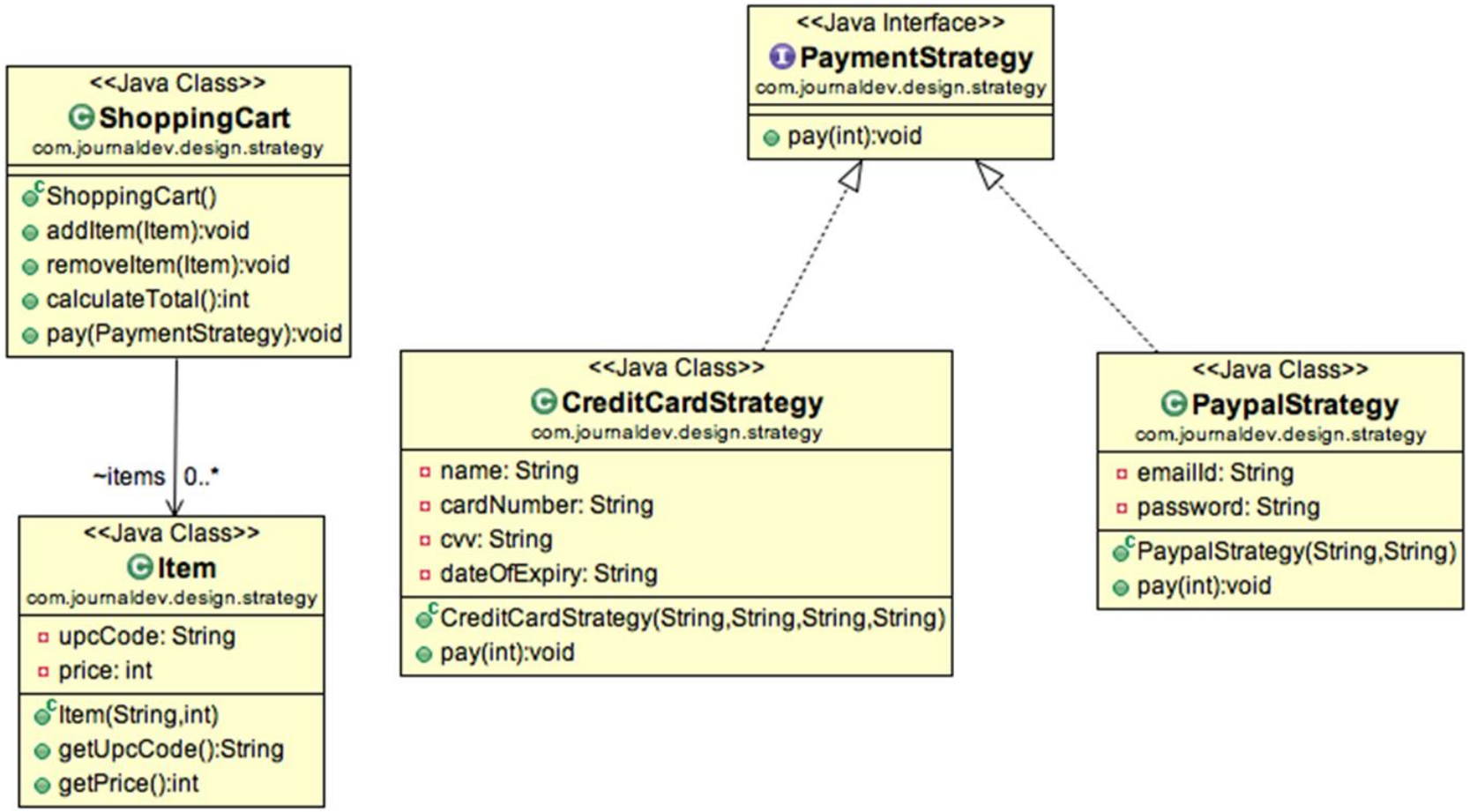

TransportationToAirport
Strategy
City bus
Personal car
Taxi

Check list…

1. Identificar un algoritmo (método) al que clientes quisieran acceder “flexiblemente”.
2. Especificar una firma para dicho algoritmo y asignarlo a una interfaz.
3. Implementar el método en una clase derivada.
4. Los clientes del algoritmo se acoplan a la interfaz.

- Ejemplos:
  - https://howtodoinjava.com/design-patterns/behavioral/strategy-design-pattern/
  - https://www.javagists.com/strategy-design-pattern

Demo


<<Java Class>>
ShoppingCart
com.journaldev.design.strategy
ShoppingCart()
addItem(Item):void
removeItem(Item):void
calculateTotal():int
pay(PaymentStrategy):void
~items
0..*
<<Java Class>>
Item
com.journaldev.design.strategy
upcCode: String
price: int
Item(String,int)
getUpcCode():String
getPrice():int
<<Java Interface>>
PaymentStrategy
com.journaldev.design.strategy
pay(int):void
<<Java Class>>
CreditCardStrategy
com.journaldev.design.strategy
name: String
cardNumber: String
cvv: String
dateOfExpiry: String
CreditCardStrategy(String,String,String,String)
pay(int):void
<<Java Class>>
PaypalStrategy
com.journaldev.design.strategy
emailId: String
password: String
PaypalStrategy(String,String)
pay(int):void

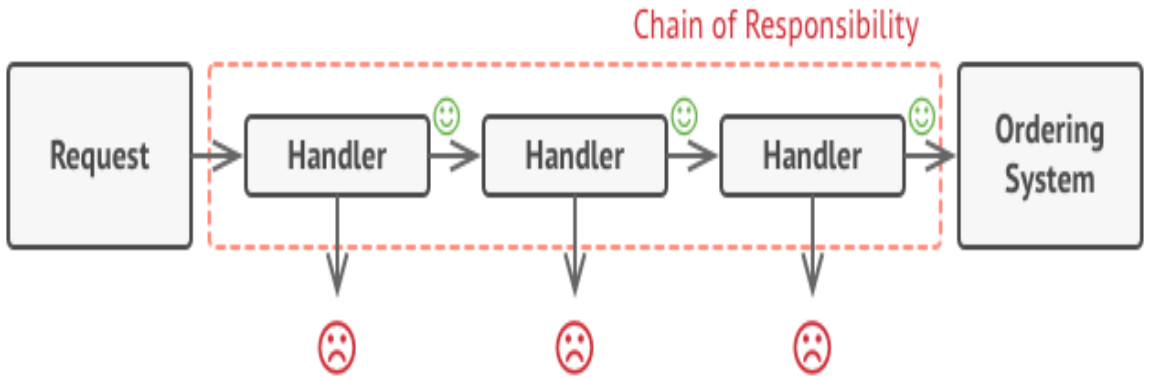
Participación

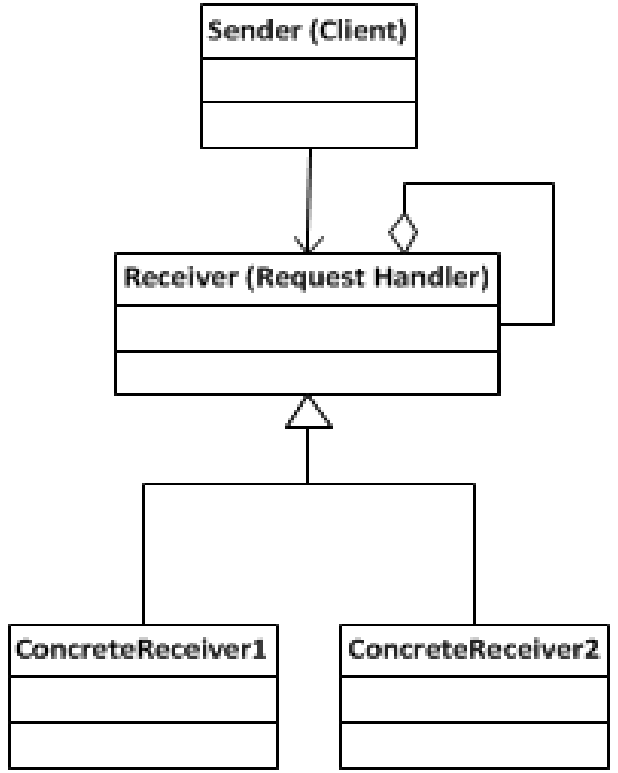
- Realice un diagrama de clases para el proceso de autenticación de LMS Sidweb y responder la actividad **Estrategia de autenticación**.

Chain of Responsibility

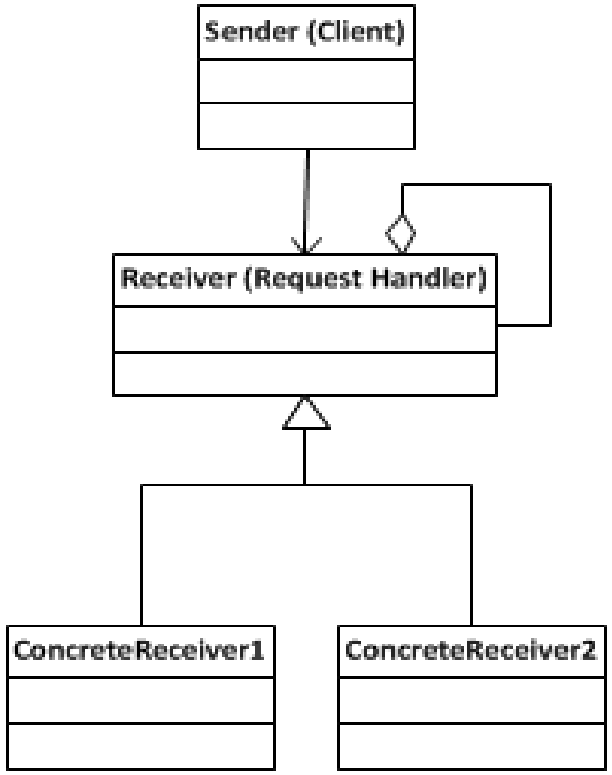
Chain of Responsibility

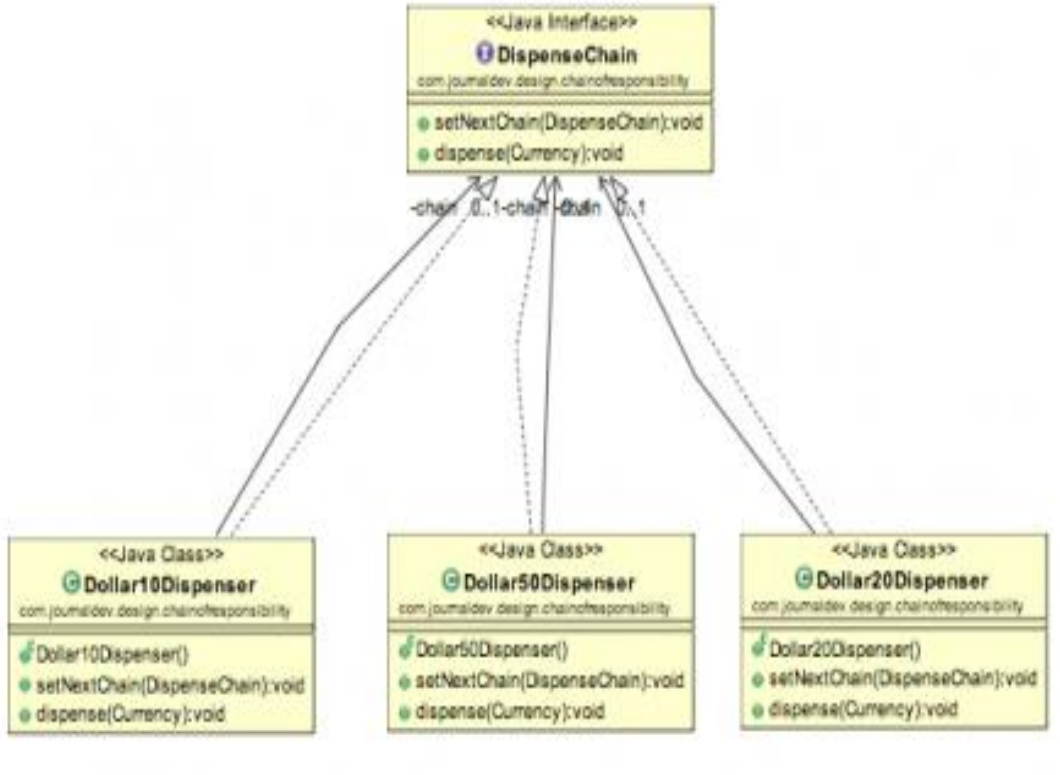
- Hay una cadena de manejadores y uno o varios de ellos puedes resolver el requerimiento o parte del mismo.
- Ejemplos:
  - Entrega de dinero en ATM
  - Manejo de Excepciones





Cajero electrónico - ATM





https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java

- Ejemplo:
  - https://codenuclear.com/chain-of-responsibility-design-pattern-in-java/

Antes de finalizar

Puntos para recordar

- ¿Qué es un patrón de diseño de comportamiento?
  - ¿Cuál es la lista de patrones dentro de esta categoría?
  - Defina la utilidad de cada patrón de comportamiento en sus propias palabras
- Asocie un ejemplo que le resulte fácil de recordar para cada patrón de diseño

Lectura adicional

- Gamma et al. , “Design Patterns: Elements of Reusable Object-Oriented Software”
- Shalloway and Trott, "Design Patterns Explained"
- Source Making, “Design Patterns”
  - https://sourcemaking.com/design_patterns

Próxima sesión

- Refactoring