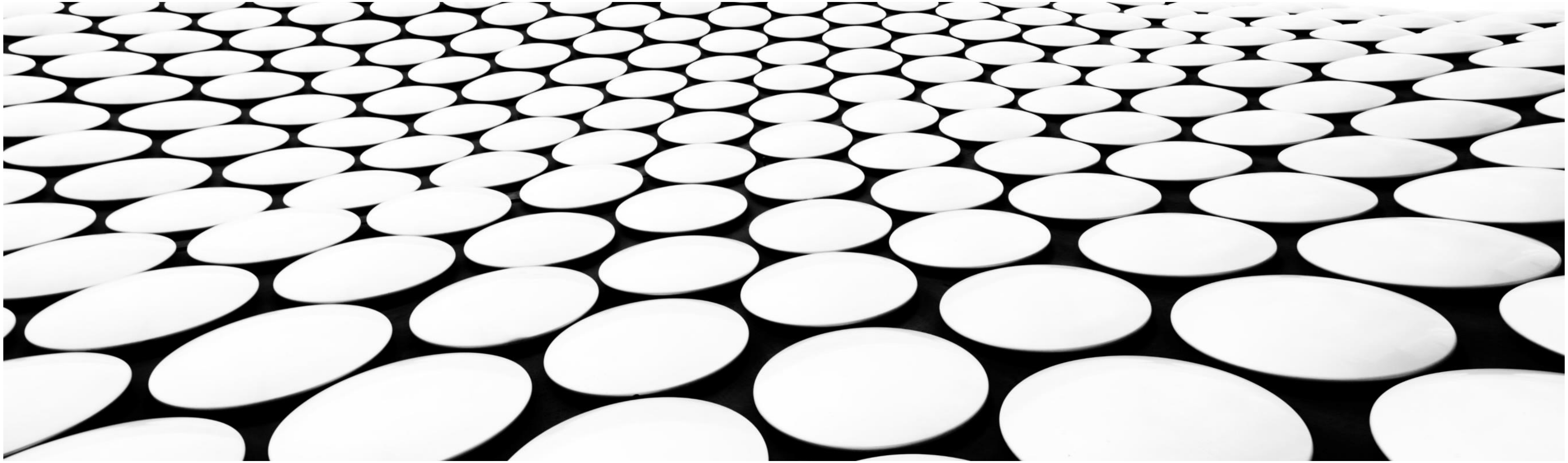
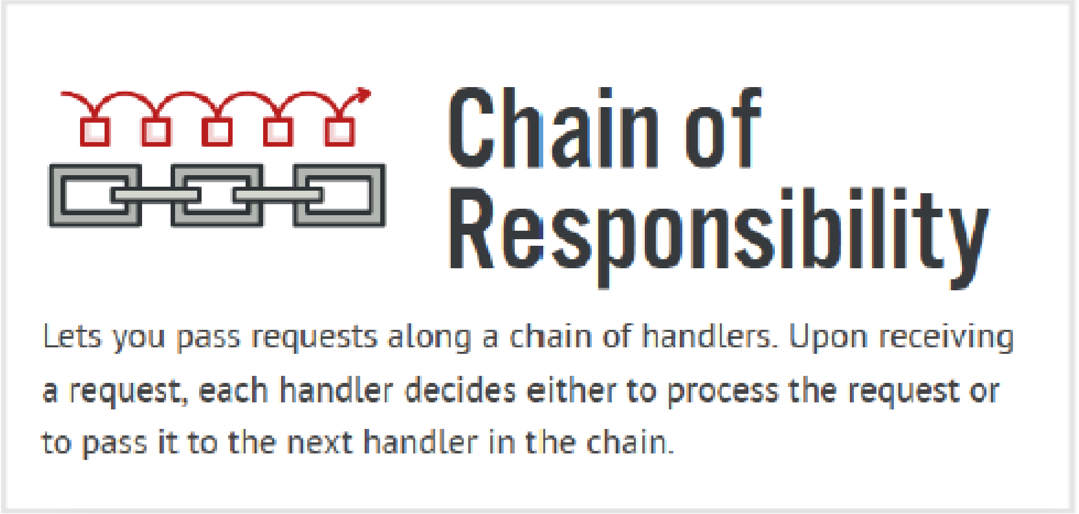
BEHAVIORAL DESIGN PATTERNS

OBSERVER , STRATEGY, COMMAND



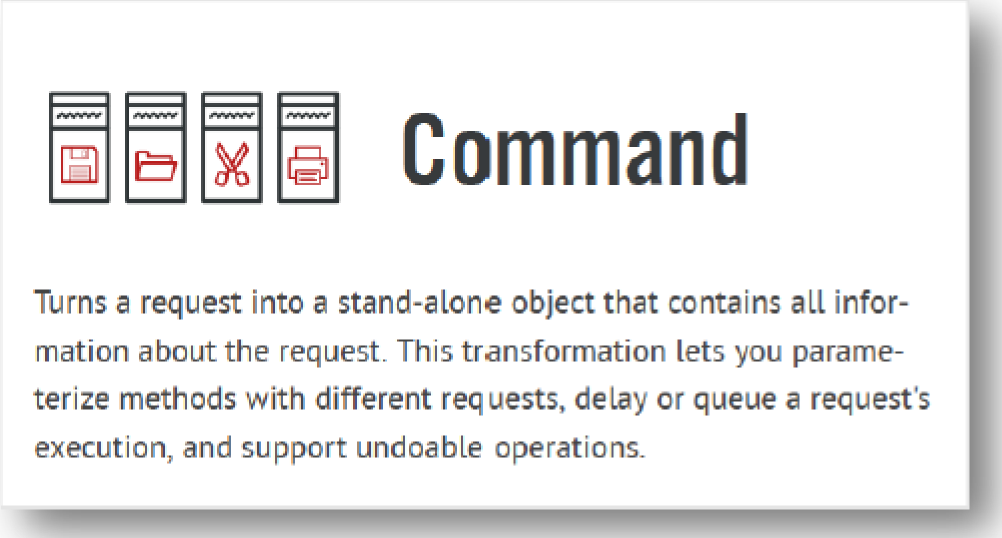
WHAT IS A *BEHAVIORAL* PATTERN?

- Help you define the communication between objects in your system and how the flow is controlled in a complex program.
- Behavior patterns are concerned with algorithms and the assignment of responsibilities between objects.
- The *Behavioral* design patterns includes :



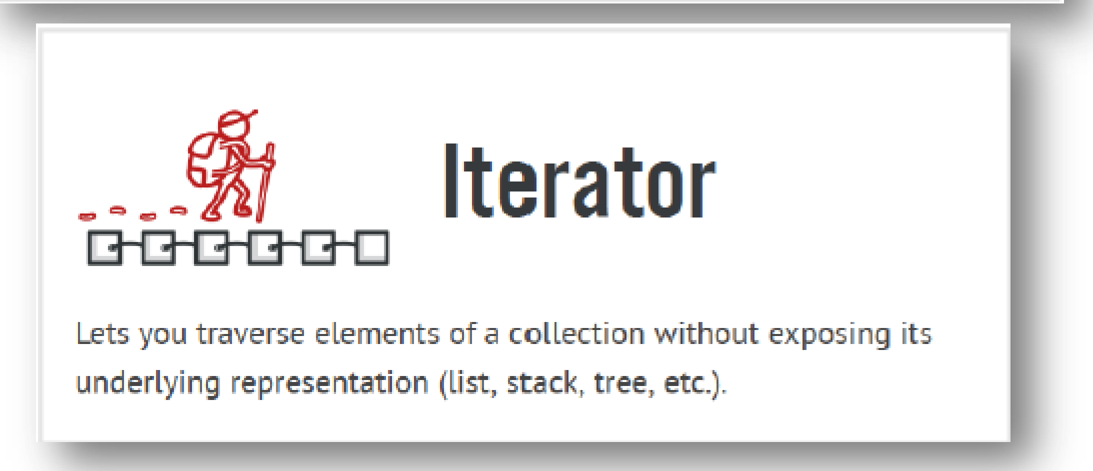
Chain of Responsibility

Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.



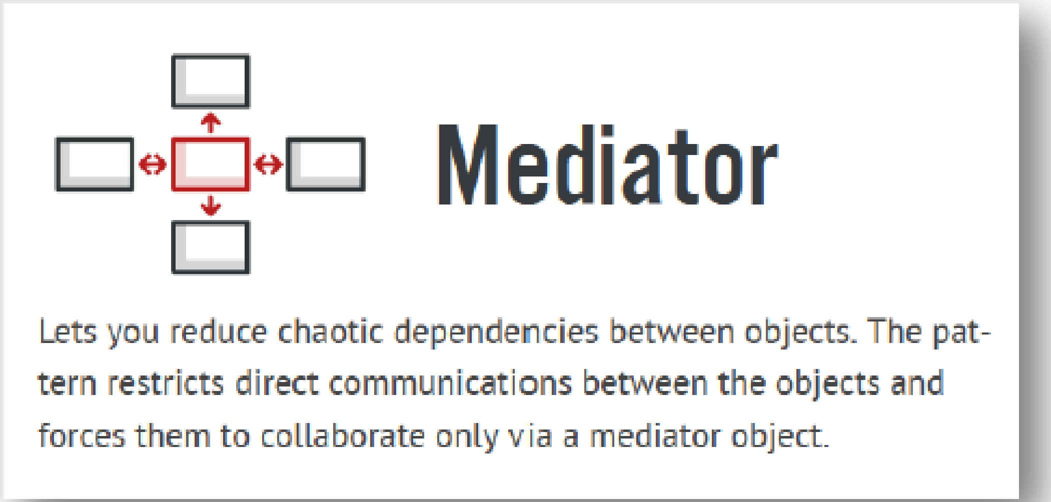
Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.



Iterator

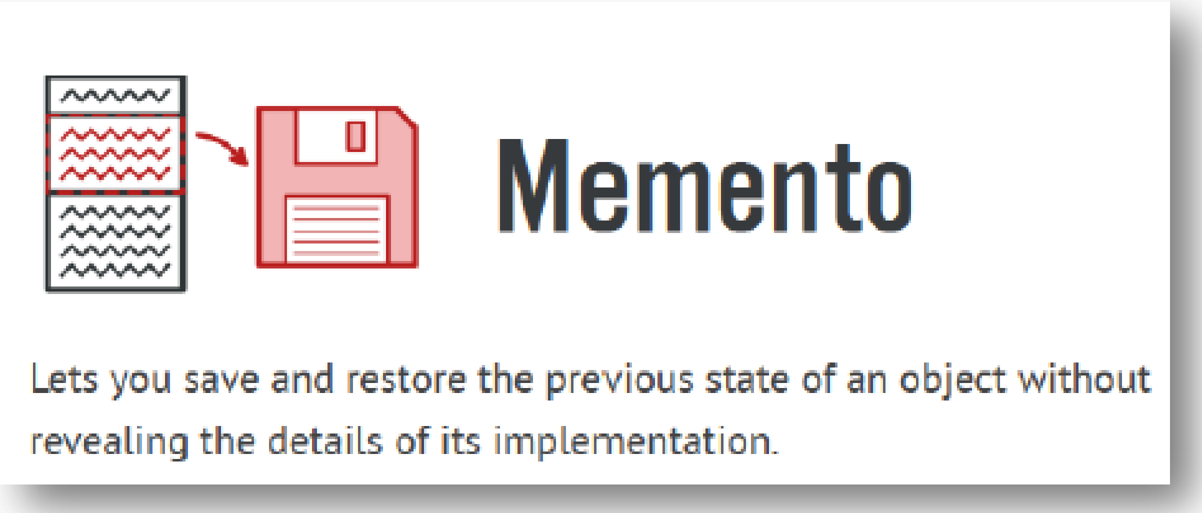
Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



Mediator

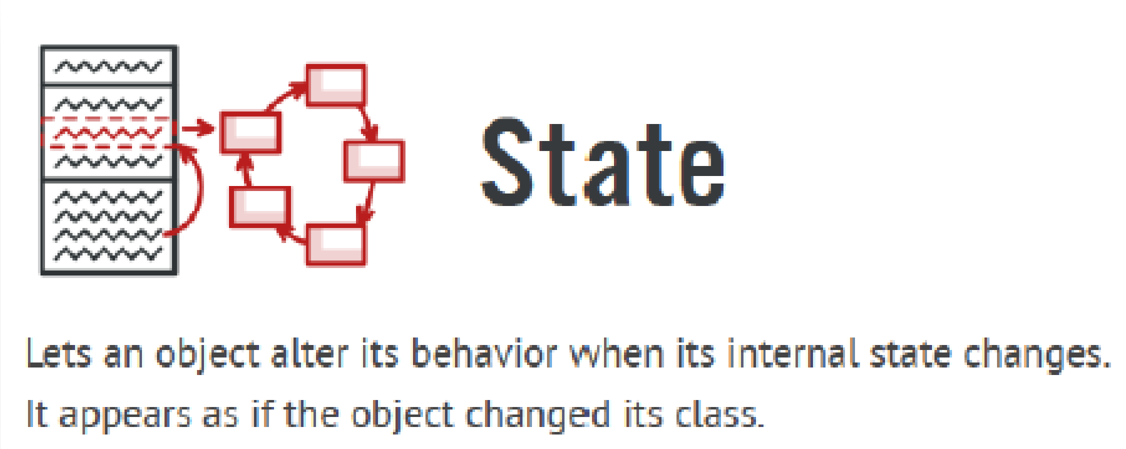
Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

WHAT IS A *BEHAVIORAL* PATTERN CONT ...?



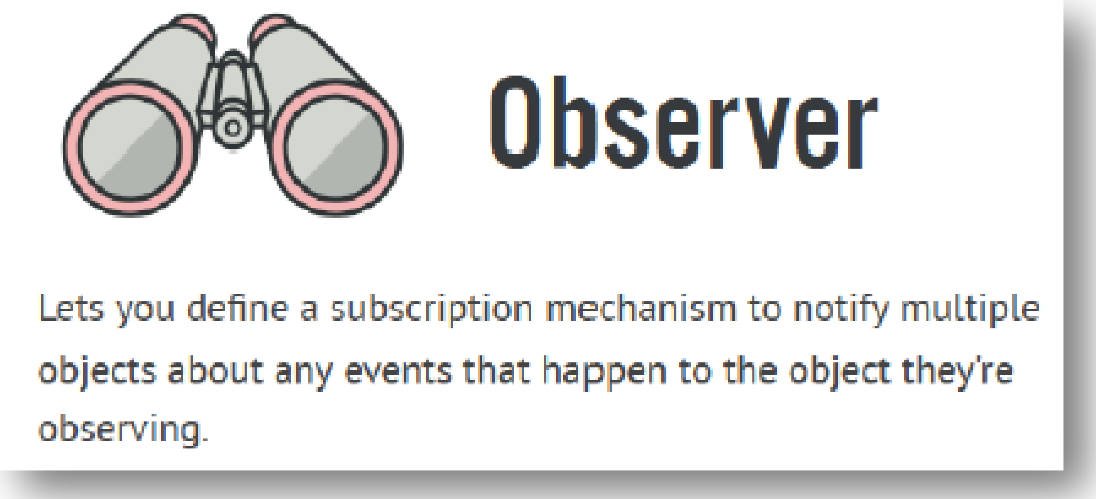
Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.



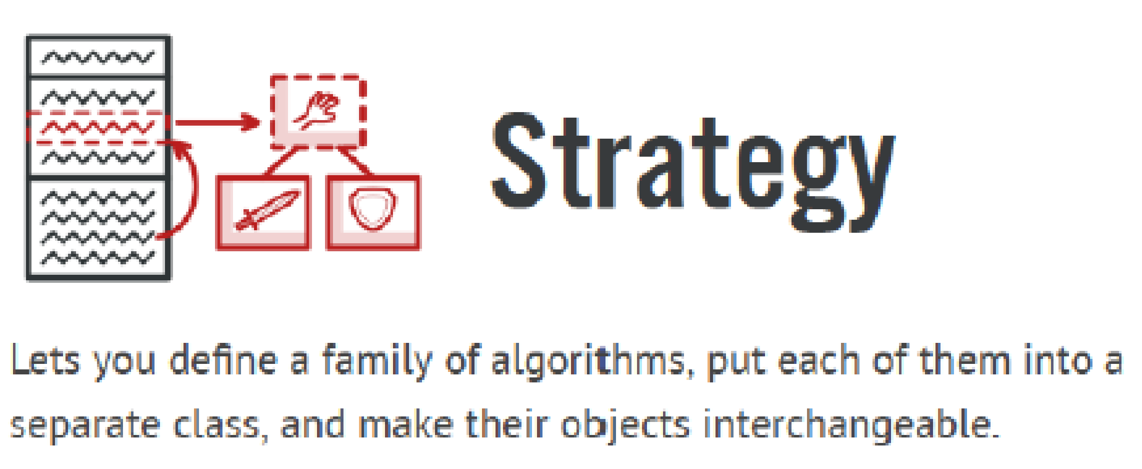
State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



Observer

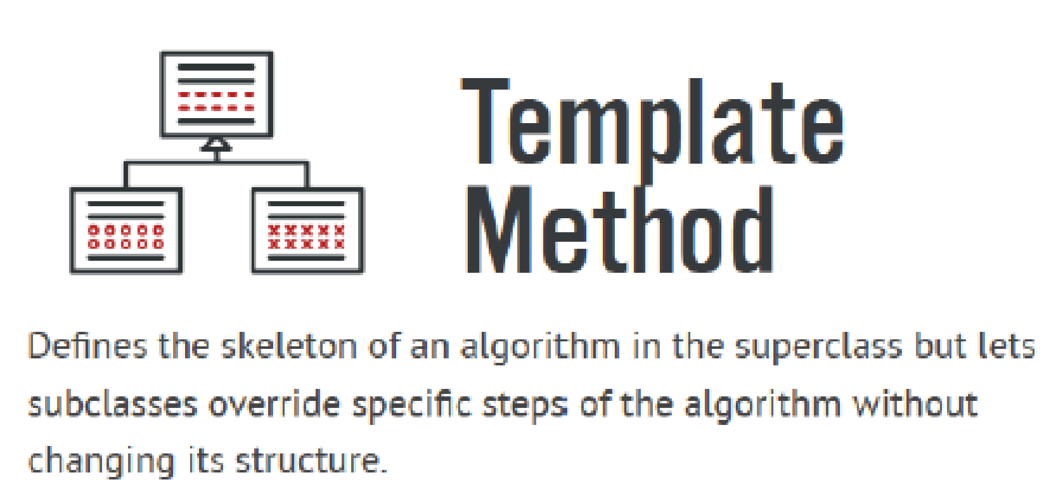
Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



Strategy

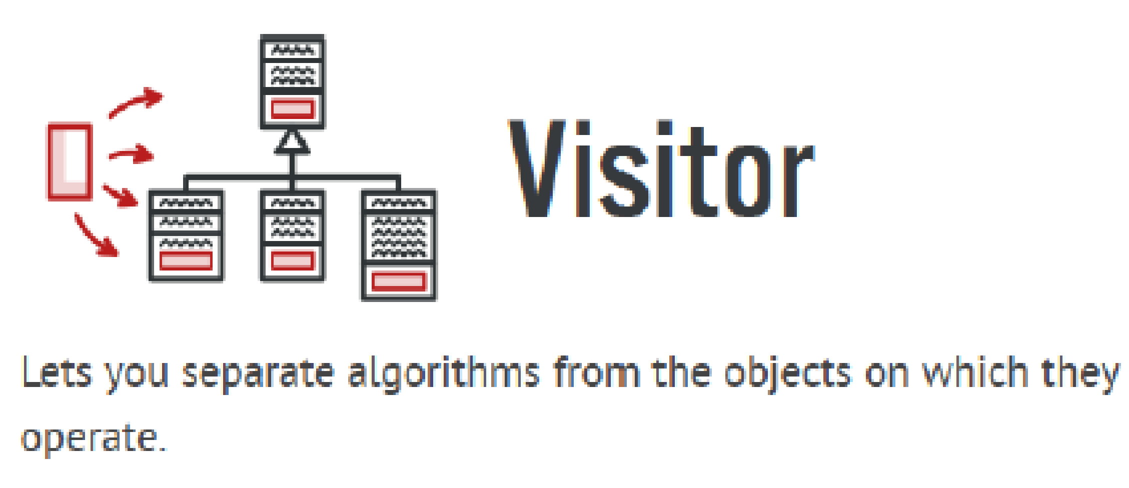
Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

WHAT IS A *BEHAVIORAL* PATTERN CONT …?



Template Method

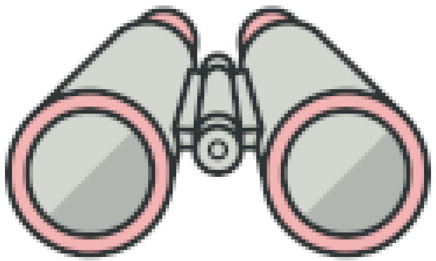
Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.



Visitor

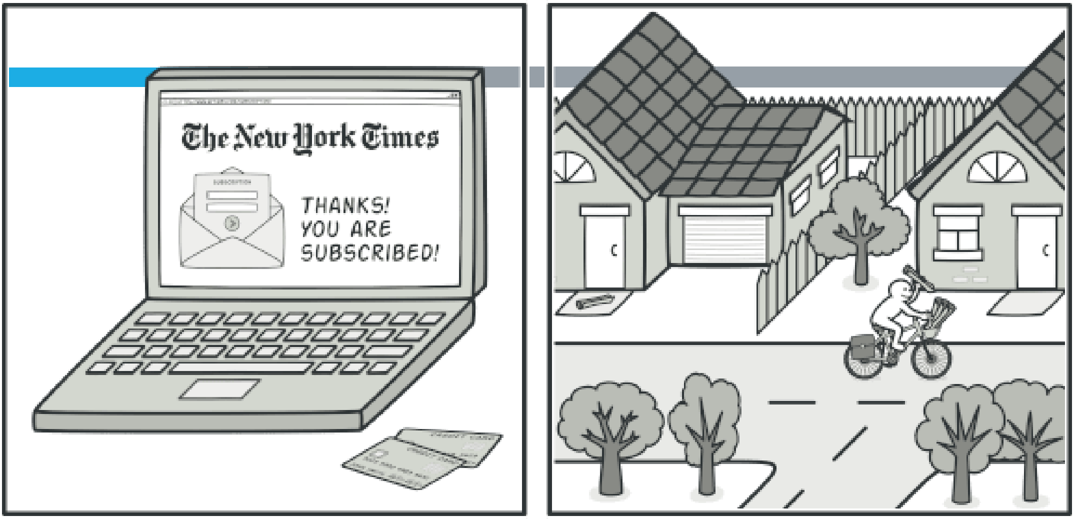
Lets you separate algorithms from the objects on which they operate.

OBSERVER DESIGN PATTERN



- *Also known as: **Event-Subscriber, Listener***
- **Observer** is a behavioral design pattern that allows some objects to notify other objects about changes in their state.
- “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”

CONCEPT



- Observers are basically interested and want to be notified when there is a change made inside that subject. So, they register themselves to that subject. When they lose interest in the subject they simply unregister from the subject.
- Sometimes this model is also referred to as the **Publisher-Subscriber model.**
- Magazine and newspaper subscriptions.
    - If you subscribe to a newspaper or magazine, you no longer need to go to the store to check if the next issue is available. Instead, the publisher sends new issues directly to your mailbox right after publication or even in advance.
    - The publisher maintains a list of subscribers and knows which magazines they're interested in. Subscribers can leave the list at any time when they wish to stop the publisher sending new magazine issues to them.

REAL-LIFE EXAMPLES

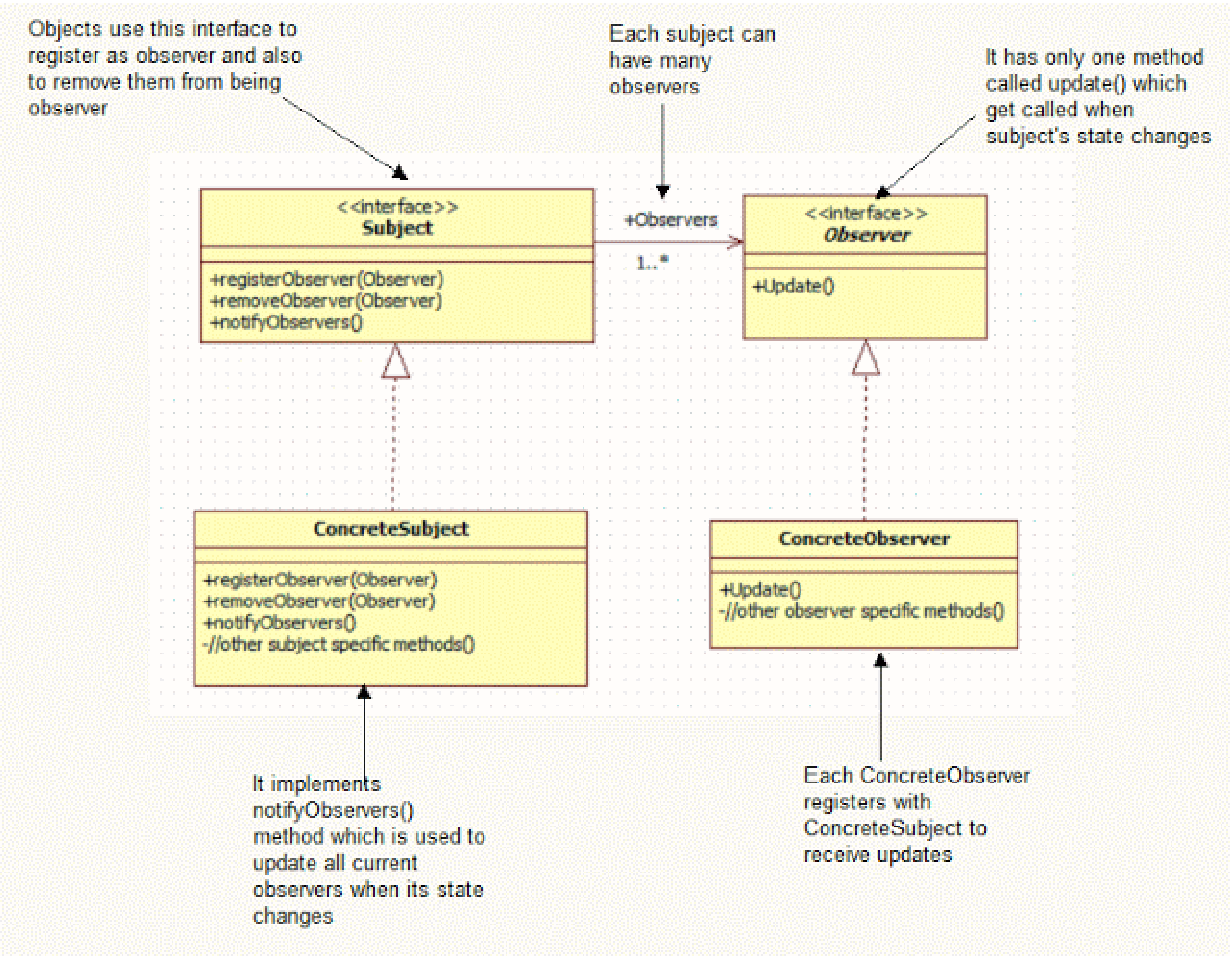
- You might have surfed “Flipkart.com-Online megastore". So when you search for any product and it is unavailable then there is option called “Notify me when product is available”. If you subscribe to that option then when state of product changes i.e. it is available, you will get notification mail “Product is available now you can buy it". In this case, Product is subject and You are an observer.
- Lets say, your permanent address is changed then you need to notify passport authority and pan card authority. So here passport authority and pan card authority are observers and You are a subject.
- On Facebook also, If you subscribe someone then whenever new updates happen then you will be notified.

WHEN TO USE IT:

- When one object changes its state, then all other dependents object must automatically change their state to maintain consistency
- When subject doesn’t know about number of observers it has.
- When an object should be able to notify other objects without knowing who objects are.

UML & PARTICIPANTS

- Subject
  - Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.
- Observer
  - defines an updating interface for objects that should be notified of changes in a subject.
- ConcreteSubject
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- ConcreteObserver
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistent with the subject's.
  - implements the Observer updating interface to keep its state consistent with the subject's.


<<interface>>
Subject
+registerObserver(Observer)
+removeObserver(Observer)
+notifyObservers()
+Observers
1..*
<<interface>>
Observer
+Update()
ConcreteSubject
+registerObserver(Observer)
+removeObserver(Observer)
+notifyObservers()
-//other subject specific methods()
ConcreteObserver
+Update()
-//other observer specific methods()

Objects use this interface to register as observer and also to remove them from being observer

Each subject can have many observers

It has only one method called update() which get called when subject's state changes

It implements notifyObservers() method which is used to update all current observers when its state changes

Each ConcreteObserver registers with ConcreteSubject to receive updates

TWEET NOTIFICATION EXAMPLE USING OBSERVER DESIGN PATTERN

- First we will define the **Subject** interface:

```
public interface Subject {
    public void addSubscriber(Observer
observer);
    public void removeSubscriber(Observer
observer);
    public void notifySubscribers(String tweet);

}
```

- Then we will create **Observer** interface:

```
public interface Observer {
    public void notification(String handle, String
tweet);
}
```

- Then we will create concrete subject class called **PublicFigure**

```
import java.util.ArrayList;
import java.util.List;
public class PublicFigure implements Subject {
protected List<Observer> observers = new ArrayList<Observer>();
   protected String name;
   protected String handle;
  public PublicFigure(String name, String handle) {
      super();
      this.name = name;
      this.handle = "#" + handle;
   }
   public String getName() {
      return name;
   }
  public void setName(String name) {
      this.name = name;
   }
  public String getHandle() {
      return handle;
   }
```

```
 public void tweet(String tweet) {
   System.out.printf("\nName: %s, Tweet: %s\n", name, tweet);
      notifySubscribers(tweet);
   }
@Override
   public synchronized void addSubscriber(Observer observer) {
      observers.add(observer);
   }
@Override
public synchronized void removeSubscriber(Observer observer) {
observers.remove(observer);
}
@Override
   public void notifySubscribers(String tweet) {
    observers.forEach(observer -> observer.notification(handle,
tweet));
   }    }
```

- Now we will define **Follower** class:

```
public class Follower implements Observer {
protected String name;

public Follower(String name) {
super();
        this.name = name;
    }

@Override
    public void notification(String handle, String tweet) {
        System.out.printf("'%s' received notification from Handle:
'%s', Tweet: '%s'\n", name, handle, tweet);
}
}
```

```
public class Main {

public static void main(String args[]) {
  PublicFigure bobama = new PublicFigure("Barack Obama",
"bobama");
  Follower ajay = new Follower("Ajay");
  Follower vijay = new Follower("Vijay");
  Follower racheal = new Follower("Racheal");
  Follower micheal = new Follower("Micheal");
  Follower kim = new Follower("Kim");
     bobama.addSubscriber(ajay);
     bobama.addSubscriber(vijay);
     bobama.addSubscriber(racheal);
     bobama.addSubscriber(micheal);
     bobama.addSubscriber(kim);

    bobama.tweet("Hello Friends!");
     bobama.removeSubscriber(racheal);
     bobama.tweet("Stay Home! Stay Safe!");
}   }
```

- And here's the output:

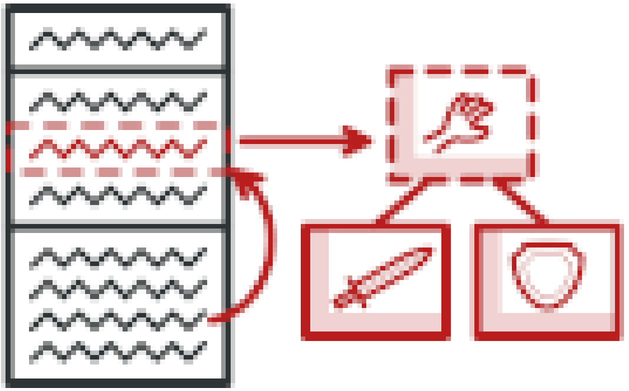
```
Name: Barack Obama, Tweet: Hello Friends!
'Ajay' received notification from Handle: '#bobama', Tweet: 'Hello Friends!'
'Vijay' received notification from Handle: '#bobama', Tweet: 'Hello Friends!'
'Racheal' received notification from Handle: '#bobama', Tweet: 'Hello Friends!'
'Micheal' received notification from Handle: '#bobama', Tweet: 'Hello Friends!'
'Kim' received notification from Handle: '#bobama', Tweet: 'Hello Friends!'


Name: Barack Obama, Tweet: Stay Home! Stay Safe!
'Ajay' received notification from Handle: '#bobama', Tweet: 'Stay Home! Stay
Safe!'
'Vijay' received notification from Handle: '#bobama', Tweet: 'Stay Home! Stay
Safe!'
'Micheal' received notification from Handle: '#bobama', Tweet: 'Stay Home!
Stay Safe!'
'Kim' received notification from Handle: '#bobama', Tweet: 'Stay Home! Stay
Safe!'
```

- Please note that when I unregistered 'Racheal' from Barak Obama's follower list, she stopped receiving notifications.

PROS AND CONS

- Pros
  - *Open/Closed Principle*. You can introduce new subscriber classes without having to change the publisher's code (and vice versa if there's a publisher interface).
  - You can establish relations between objects at runtime.
- Cons
  - Subscribers are notified in random order.



STRATEGY DESIGN PATTERN

"Define a family of algorithms, encapsulate each one, and make them interchangeable. The strategy pattern lets the algorithm vary independently from client to client."

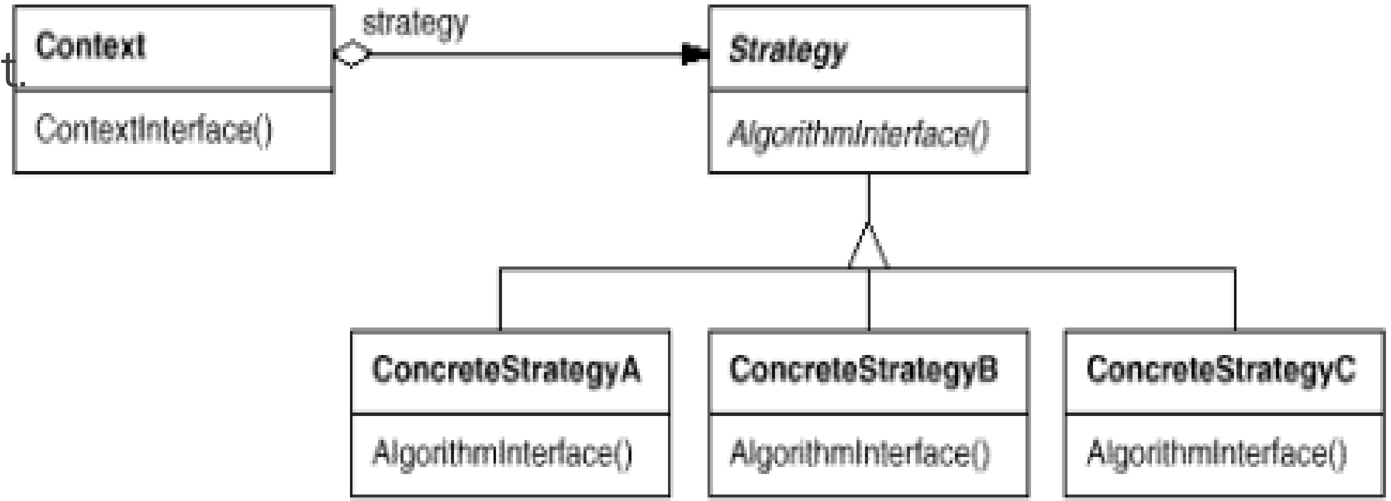
- Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
- We can select the behavior of an algorithm dynamically at runtime.

WHERE TO USE

- ❑ When you need to use one of several algorithms dynamically.
- ❑ When you want to configure a class with one of many related classes (behaviors).
- ▪ Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
- ▪ Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior.
- ▪ Use the pattern when your class has a massive conditional operator that switches between different variants of the same algorithm.

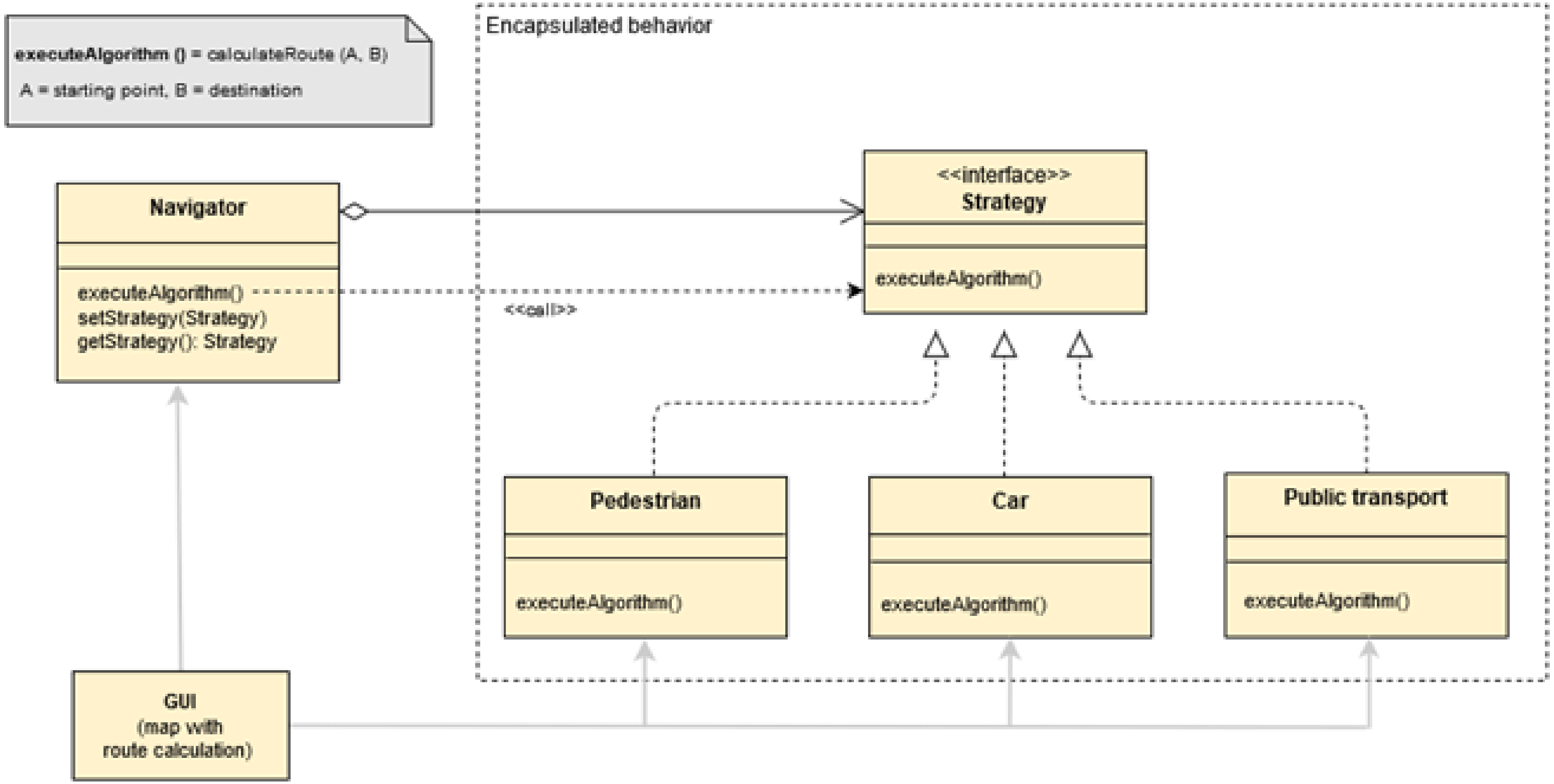
STRUCTURE & PARTICIPANTS

- Strategy
  - declares an interface common to all supported algorithms.
  - Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy
  - implements the algorithm using the Strategy interface.
- Context
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.



EXAMPLE

- In our example a navigation app is to be implemented with the help of a strategy design pattern. The **app should calculate a route** based on normal modes of transport. The user can choose between three options:
- Pedestrian (ConcreteStrategyA)
- Car (ConcreteStrategyB)
- Public transport (ConcreteStrategyC)



IN OUR EXAMPLE,

- the **client** is the graphical user interface (GUI) of a navigation app with buttons for calculating routes. Once the user makes a selection and taps on a button, a concrete route is calculated. The **Context (navigator class)** has the task of calculating and presenting a range of control points on the map. The navigator class has a method for switching the active routing strategy. This means it is possible to switch between modes of transport via the client buttons.
- For example, if the user triggers a command with the **pedestrian button of the client**, the service “Calculate the pedestrian route” (ConcreteStrategyA) is requested. The method *executeAlgorithm()* (in our example, the method: *calculateRoute (A, B)*) accepts a starting point and destination, and returns a collection of route control points. The Context accepts the client command and decides on the right strategy (*setStrategy: Pedestrian*)
- The currently selected strategy is stored in the Context (navigator class) using *getStrategy()*. The **results of the ConcreteStrategy calculations** are used in further processing and the graphical presentation of the route in the navigation app. If the user opts for a different route by clicking on the “Car” button afterwards, for example, the Context switches to the requested strategy (ConcreteStrategyB) and initiates a new calculation by means of another *call*. At the end of the process, **a modified route description is provided for travel by car**.

- Context:

```
public class Context {
   //prescribed standard value (default behavior): ConcreteStrategyA
   private Strategy strategy = new ConcreteStrategyA();
   public void execute() {
      //delegates the behavior to a Strategy object
      strategy.executeAlgorithm();
   }
   public void setStrategy(Strategy strategy) {
      strategy = strategy;
   }
   public Strategy getStrategy() {
      return strategy;
   }
}
```

Strategy, ConcreteStrategyA, ConcreteStrategyB:

```
interface Strategy {
   public void executeAlgorithm();
}
class ConcreteStrategyA implements Strategy {
   public void executeAlgorithm() {
      System.out.println("Concrete Strategy A");
   }
}
class ConcreteStrategyB implements Strategy {
   public void executeAlgorithm() {
      System.out.println("Concrete Strategy B");
   }
}
```

Client

```
public class Client {

    public static void main(String[] args) {

       Context context = new Context();
        context.execute();

        context.setStrategy(new ConcreteStrategyB());
        context.execute();
    }

}
```

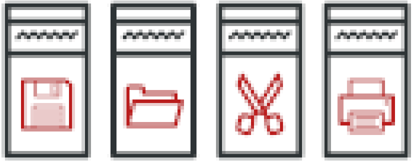
CHECK LIST

- Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
- Specify the signature for that algorithm in an interface.
- Bury the alternative implementation details in derived classes.
- Clients of the algorithm couple themselves to the interface.

PROS & CONS

- Pros
  - New algorithms that cooperate with the same interface can easily be introduced by encapsulating the algorithm separately.
  - When we have multiple algorithms for certain purposes, strategy design pattern is useful and we want our app to be flexible in selecting any of the algorithms for specific tasks.
  - Strategy design pattern allows user to select the algorithm needed without a “Switch” statement or a number of “if- else” statements.
  - You can replace inheritance with composition.
- Cons
  - If you only have a couple of algorithms and they rarely change, there’s no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
  - Clients must be aware of the differences between strategies to be able to select a proper one.

COMMAND PATTERN



- *Also known as: Action, Transaction*
- **Command** is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request’s execution, and support undoable operations.
- The definition of Command provided in the original Gang of Four book on Design Patterns states: “Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.”

CONCEPT

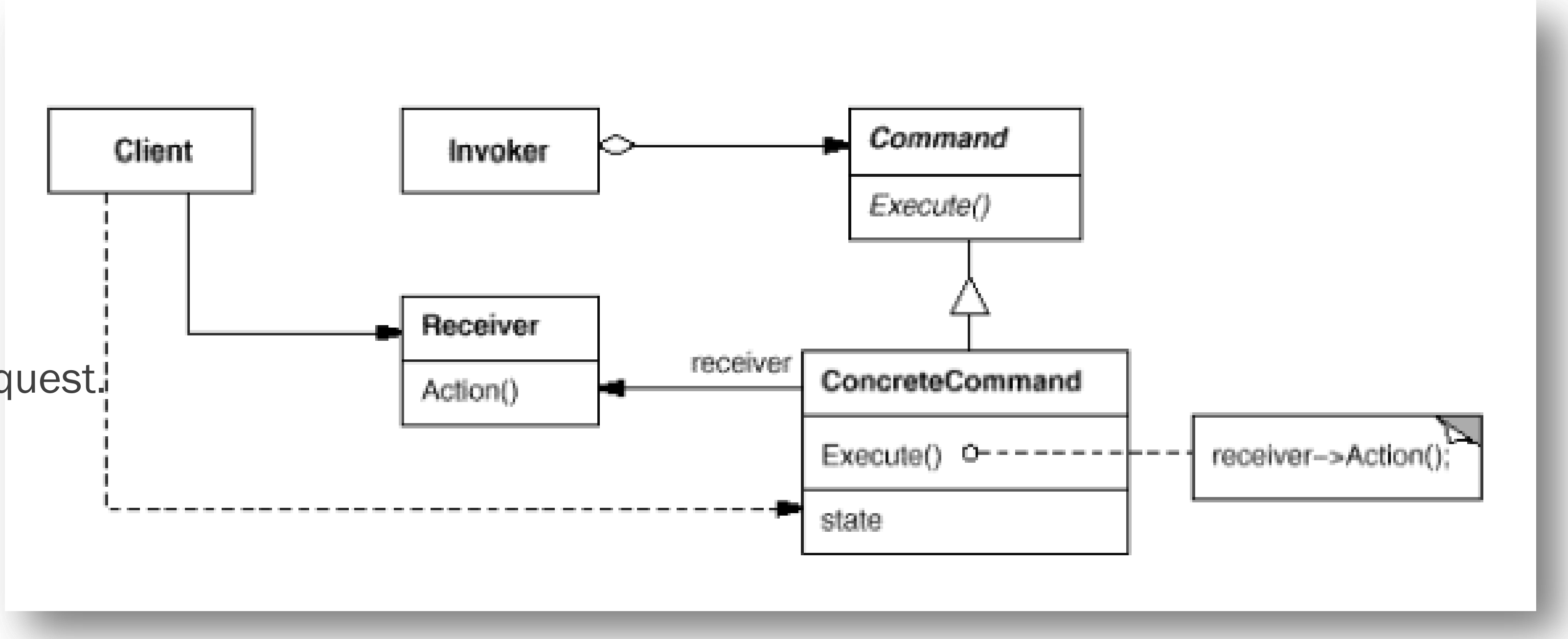
- *Command declares an interface for all commands, providing a simple execute() method which asks the Receiver of the command to carry out an operation.*
- *The Receiver has the knowledge of what to do to carry out the request.*
- *The Invoker holds a command and can get the Command to execute a request by calling the execute method.*
- *The Client creates ConcreteCommands and sets a Receiver for the command.*
- *The ConcreteCommand defines a binding between the action and the receiver. When the Invoker calls execute the ConcreteCommand will run one or more actions on the Receiver.*

WHEN TO USE IT:

- The Command Pattern is useful when:
- A history of requests is needed
- You need callback functionality
- Requests need to be handled at variant times or in variant orders
- You'll see command being used a lot when you need to have multiple undo operations, where a stack of the recently executed commands are maintained. To implement the undo, all you need to do is get the last Command in the stack and execute it's undo() method.
- You'll also find Command useful for wizards, progress bars, GUI buttons and menu actions, and other transactional behavior.

STRUCTURE & PARTICIPANTS

- Command
  - declares an interface for executing an operation.
- ConcreteCommand
  - defines a binding between a Receiver object and an action.
  - implements Execute method by invoking the corresponding operation(s) on Receiver.
- Client
  - creates a ConcreteCommand object and sets its receiver.



- Invoker
  - asks the command to carry out the request.
- Receiver
  - knows how to perform the operations.

REAL WORLD EXAMPLE

- Let's use a remote control as the example. Our remote is the center of home automation and can control everything. We'll just use a light as an example, that we can switch on or off, but we could add many more commands.
- First we'll create our command interface:

```
//Command
public interface Command{
public void execute();}
```

- Now let's create two concrete commands. One will turn on the lights, another turns off lights

```
//Concrete Command
public class LightOnCommand implements Command{
//reference to the light
Light light;
public LightOnCommand(Light light){
   this.light = light;  }
 public void execute(){
  light.switchOn();  }}
```

```
//Concrete Command
public class LightOffCommand implements Command{
//reference to the light
 Light light;
public LightOffCommand(Light light){
  this.light = light;  }
public void execute(){
 light.switchOff();  }}
```

- Light is our receiver class, so let's set that up now:

```
//Receiver
public class Light{
private boolean on;
public void switchOn(){
 on = true;  }
 public void switchOff(){
  on = false;  }}
```

- Our invoker in this case is the remote control.

```
//Invoker
public class RemoteControl{
private Command command;
 public void setCommand(Command command){
this.command = command;  }
public void pressButton(){
  command.execute();  }}
```

- Finally we'll set up a client to use the invoker

```
/Client
public class Client{
public static void main(String[] args)    {
  RemoteControl control = new RemoteControl();
 Light light = new Light();
Command lightsOn = new LightsOnCommand(light);
Command lightsOff = new LightsOffCommand(light);
//switch on
control.setCommand(lightsOn);
control.pressButton();    //switch off
control.setCommand(lightsOff);
control.pressButton();  }}
```

PROS & CONS

- Pros
  - *Single Responsibility Principle*. You can decouple classes that invoke operations from classes that perform these operations.
  - *Open/Closed Principle*. You can introduce new commands into the app without breaking existing client code.
  - You can implement undo/redo.
  - You can implement deferred execution of operations.
  - You can assemble a set of simple commands into a complex one.
- Cons
  - The code may become more complicated since you’re introducing a whole new layer between senders and receivers.

COMPARISON BETWEEN PATTERNS

- Command and Strategy may look similar because you can use both to parameterize an object with some action. However, they have very different intents.
  - You can use *Command* to convert any operation into an object. The operation’s parameters become fields of that object. The conversion lets you defer execution of the operation, queue it, store the history of commands, send commands to remote services, etc.
  - On the other hand, *Strategy* usually describes different ways of doing the same thing, letting you swap these algorithms within a single context class.
- Prototype can help when you need to save copies of Commands into history.
- **Bridge, Strategy** (and to some degree **Adapter**) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems
- **Decorator** lets you change the skin of an object, while **Strategy** lets you change the guts.
- **Command**, and **Observer** address various ways of connecting senders and receivers of requests:
  - *Command* establishes unidirectional connections between senders and receivers.
  - *Observer* lets receivers dynamically subscribe to and unsubscribe from receiving requests.

THE DIFFERENCE BETWEEN BRIDGE AND ABSTRACT FACTORY

- **Abstract Factory** is *creational design pattern*, which deals with object creation. **Bridge** is *structural design pattern*, which deals with class structure and composition.
- In **Bridge**, abstraction and implementation will vary independently. But in abstract factory, if you change abstraction (interface), you have to change client.
- In **Bridge**, The class itself can be considered as the implementation and the behavior of the class as the abstraction.
- The Abstract Factory on the other hand provides an interface for creating groups of related or dependent objects,