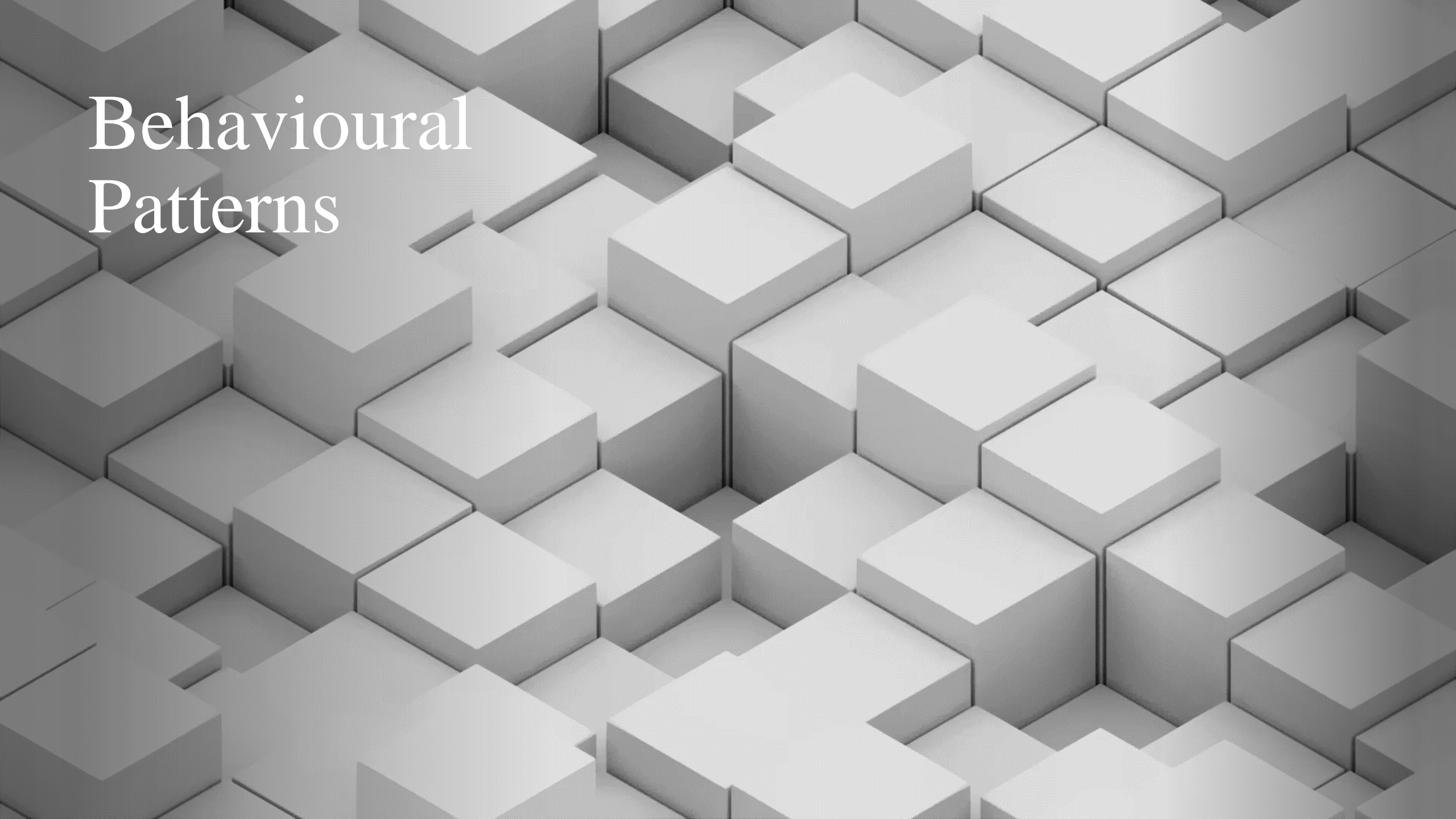
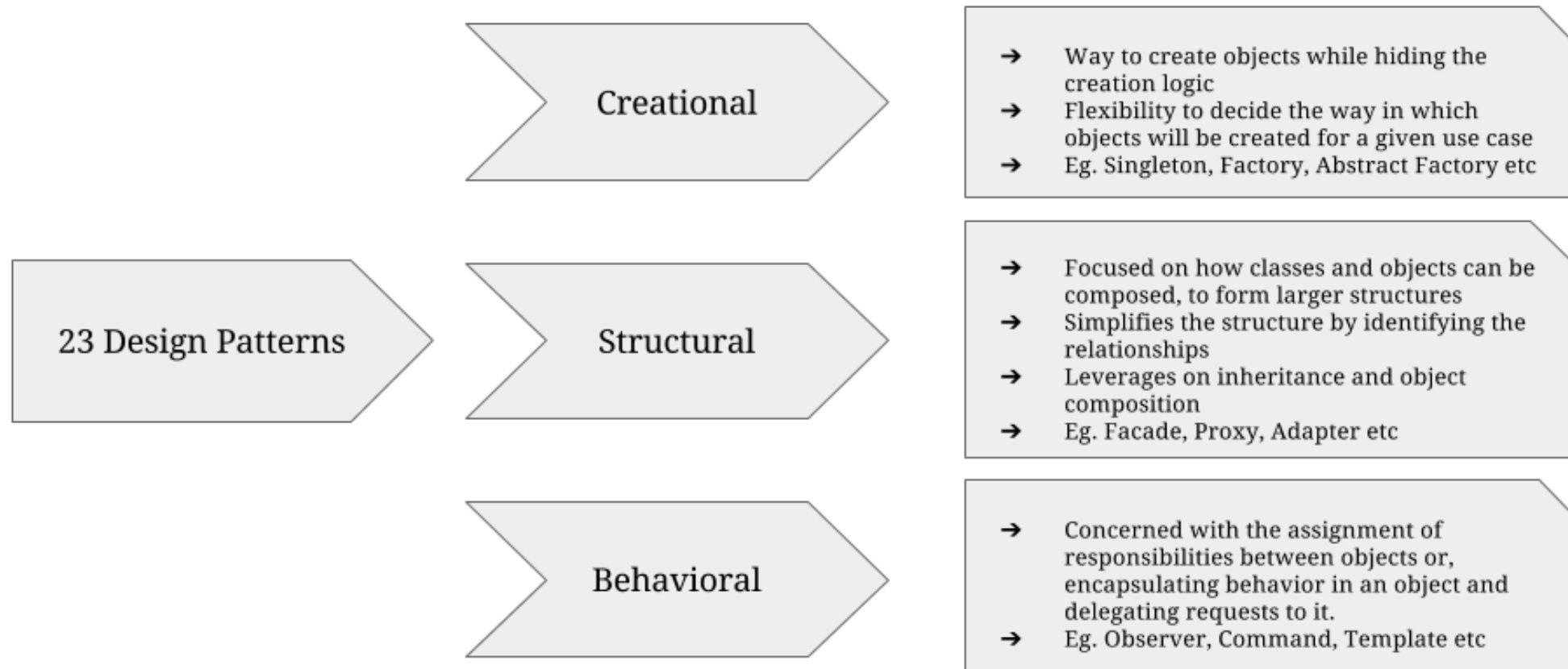


Behavioural Patterns

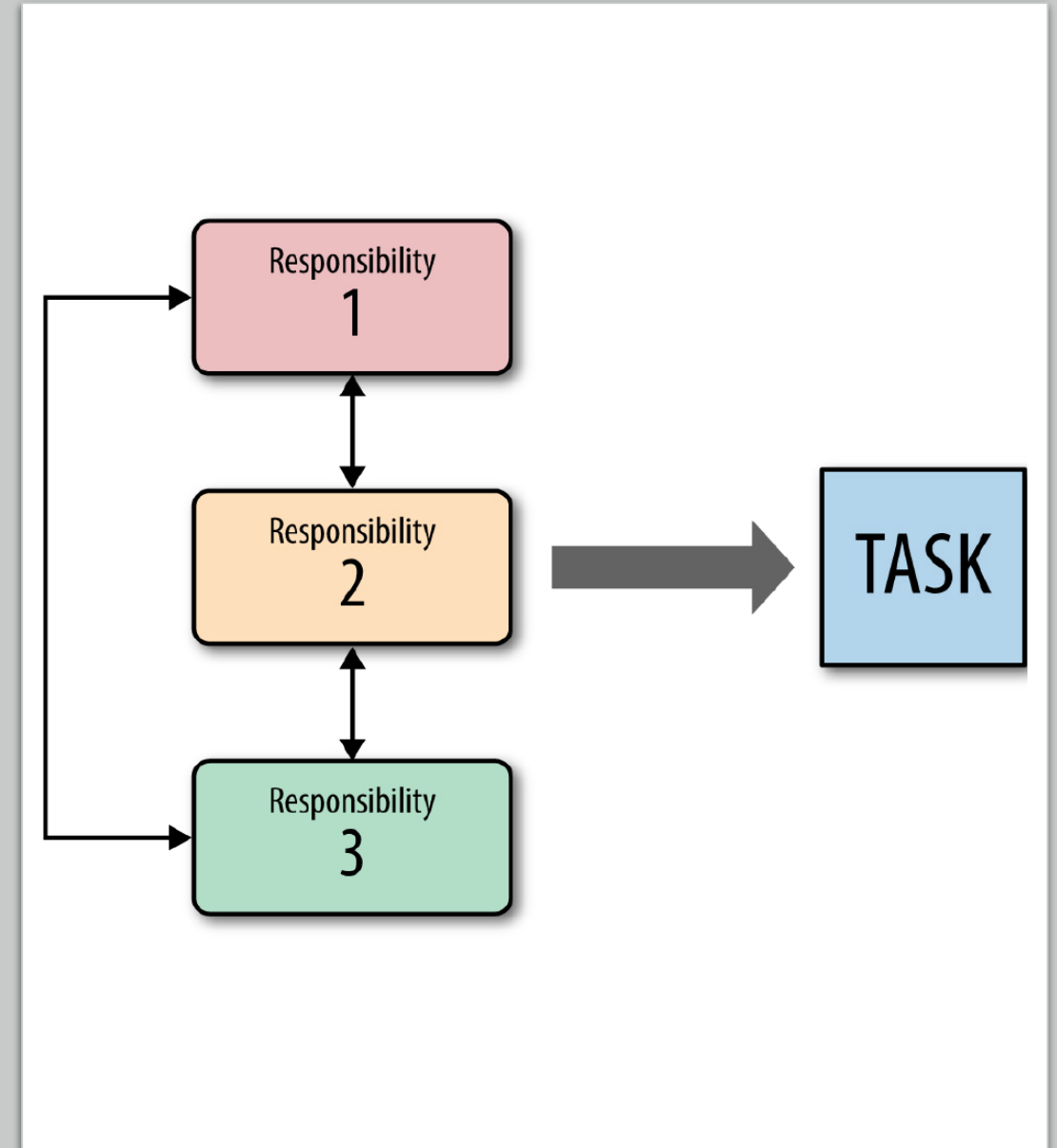


DESIGN PATTERN CLASSIFICATION & USAGE



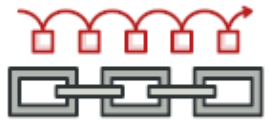
Behavioural Pattern

- The behavioral design patterns, patterns that describe the way objects and classes interact and divide responsibilities among themselves.
- A *behavioral pattern* abstracts an action you want to take from the object or class that takes the action.
- By changing the object or class, you can change the algorithm used, the objects affected, or the behavior, while still retaining the same basic interface for client classes.
- Behavioral object patterns use object composition rather than inheritance.



Behavioural Pattern

Behavioural design patterns are concerned with algorithms and the assignment of responsibilities between objects.



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.



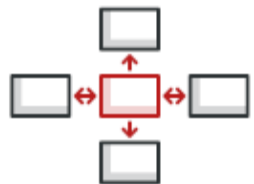
Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.



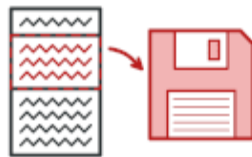
Iterator

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



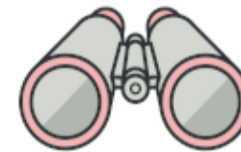
Mediator

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.



Memento

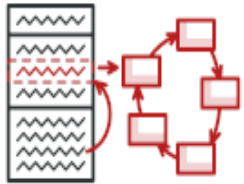
Lets you save and restore the previous state of an object without revealing the details of its implementation.



Observer

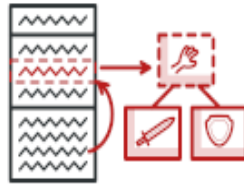
Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Behavioural Pattern



State

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



Strategy

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



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

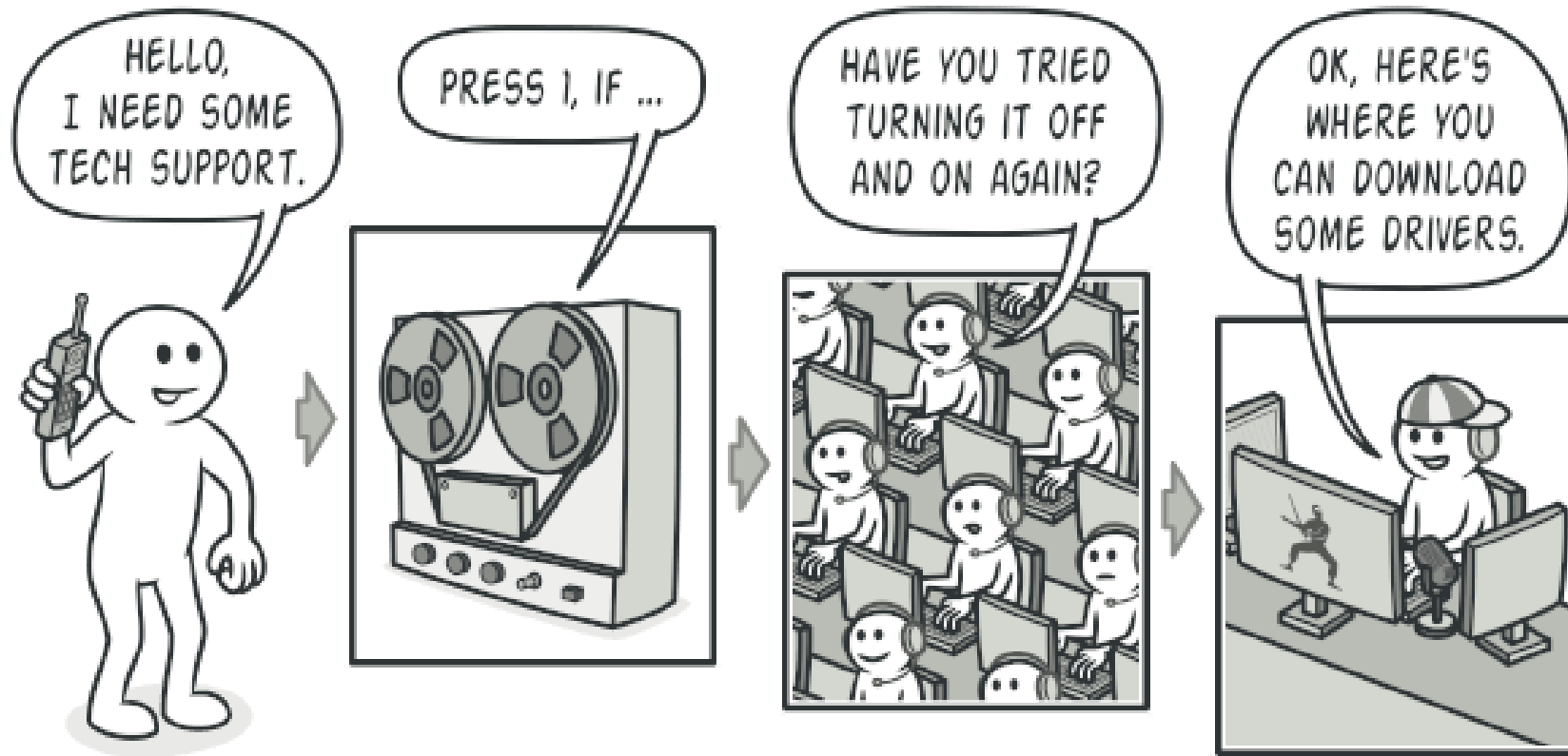
Lets you separate algorithms from the objects on which they operate.

Chain Of Responsibility Pattern

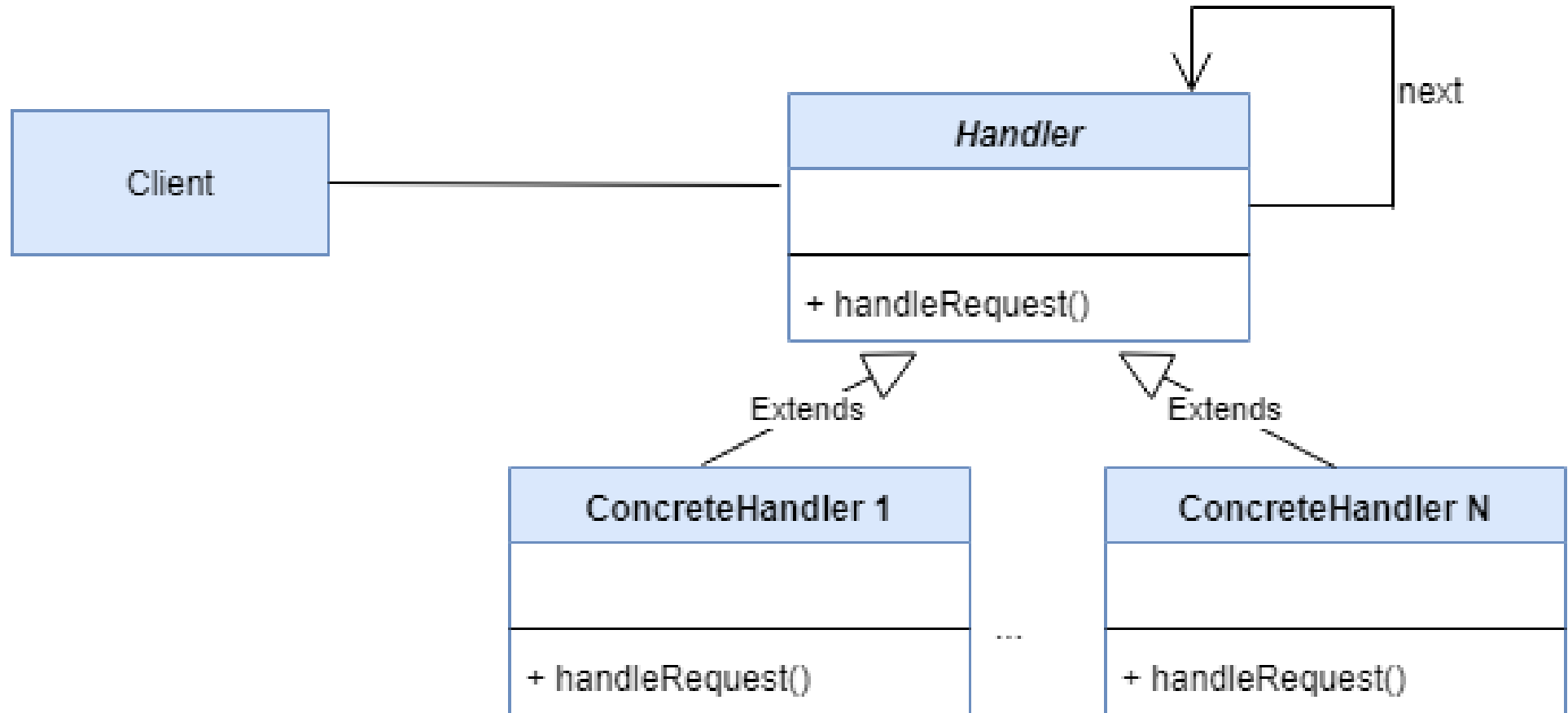
The logic defined by GoF is :

"Gives more than one object an opportunity to handle a request by linking receiving objects together."

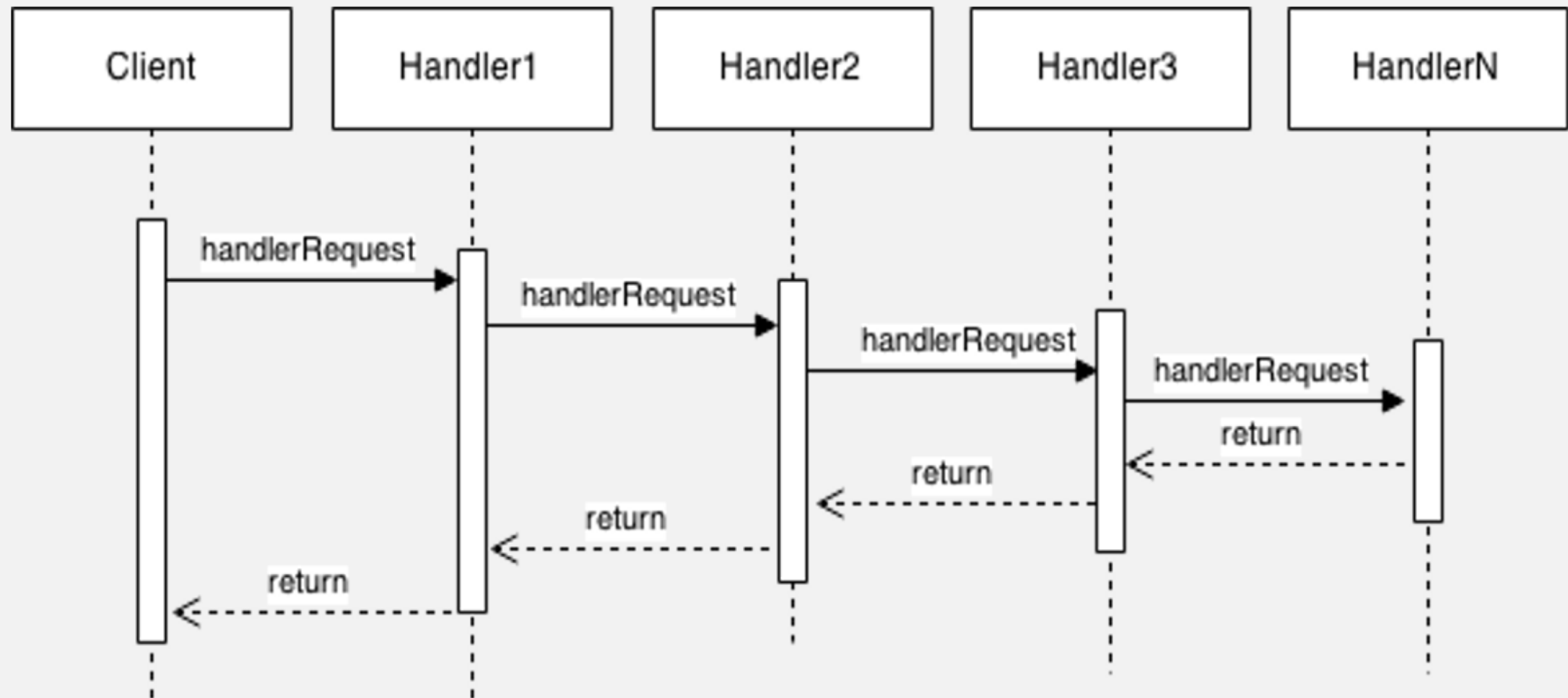
Chain Of Responsibility Pattern



Chain of Responsibility Pattern UML Representation:



Chain of Responsibility pattern – Diagram of sequence



Chain Of Responsibility Pattern

- When there are more than one objects that can handle or fulfill a client request, the pattern recommends giving each of these objects a chance to process the request in some sequential order.
- Applying the pattern in such a case, each of these potential handlers can be arranged in the form of a chain, with each object having a reference to the next object in the chain.
- The first object in the chain receives the request and decides either to handle the request or to pass it on to the next object in the chain.
- The request flows through all objects in the chain one after the other until the request is handled by one of the handlers in the chain or the request reaches the end of the chain without getting processed.

Chain Of Responsibility Pattern

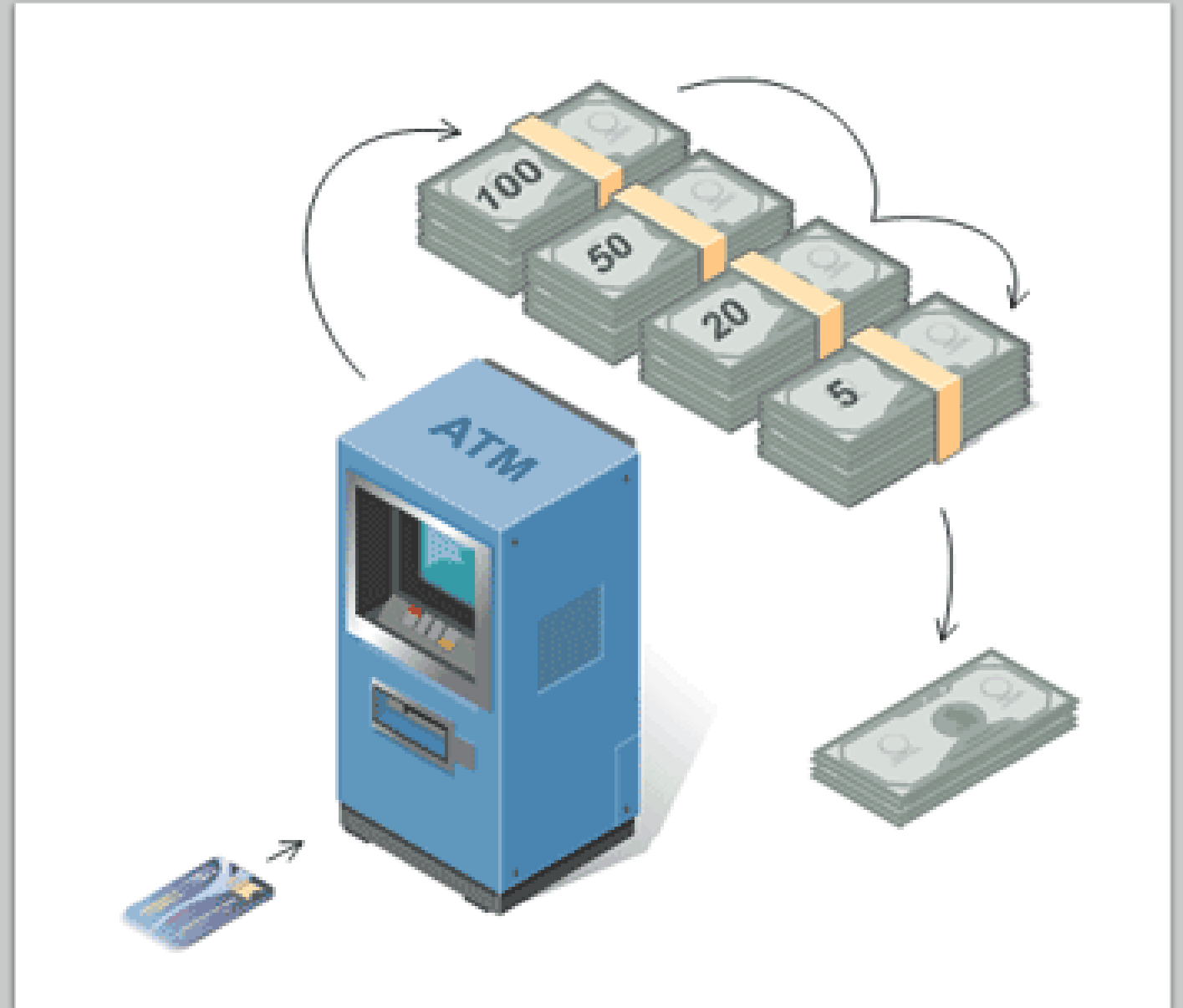
Example 1



Chain Of Responsibility Pattern

Example 2

- ATM money dispenser
- In ATM Dispense machine, the user enters the amount to be dispensed and the machine dispense amount in terms of defined currency bills such as 50\$, 20\$, 10\$ etc. If the user enters an amount that is not multiples of 10, it throws error.



When to use the Chain of Responsibility Pattern

1. More than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
2. You want to issue a request to one of several objects without specifying the receiver explicitly.
3. The set of objects that can handle a request should be specified dynamically.
4. These objects & its order determined at run time on the basis of request type.

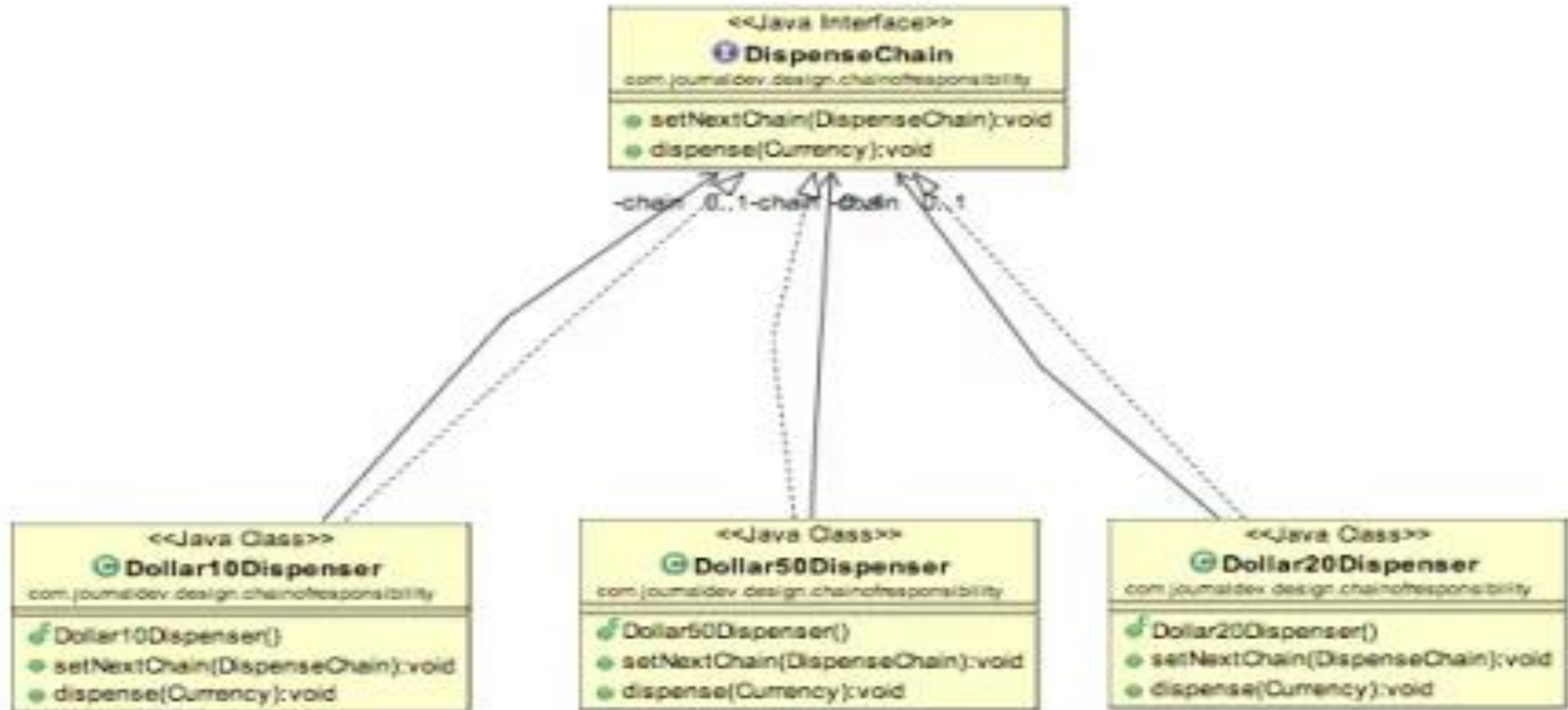
Chain of Responsibility Pattern Examples in JDK

- `util.logging.Logger#log ()`
- `servlet.Filter#doFilter ()`

Pros and Cons

- You can control the order of request handling.
- *Single Responsibility Principle*. You can decouple classes that invoke operations from classes that perform operations.
- *Open/Closed Principle*. You can introduce new handlers into the app without breaking the existing client code.
- Some requests may end up unhandled.
- It can create deep stack traces, which can affect performance.
- It can lead to duplicate code across processors, increasing maintenance.

Chain of Responsibility



Chain of Responsibility

Let us consider the transaction approval process in a company. Suppose we want to approve the transactions based on certain conditions?

For instance, in the transaction approval application user keys in transaction details into the application and this transaction need to be processed by any one of the higher level employee in the company.

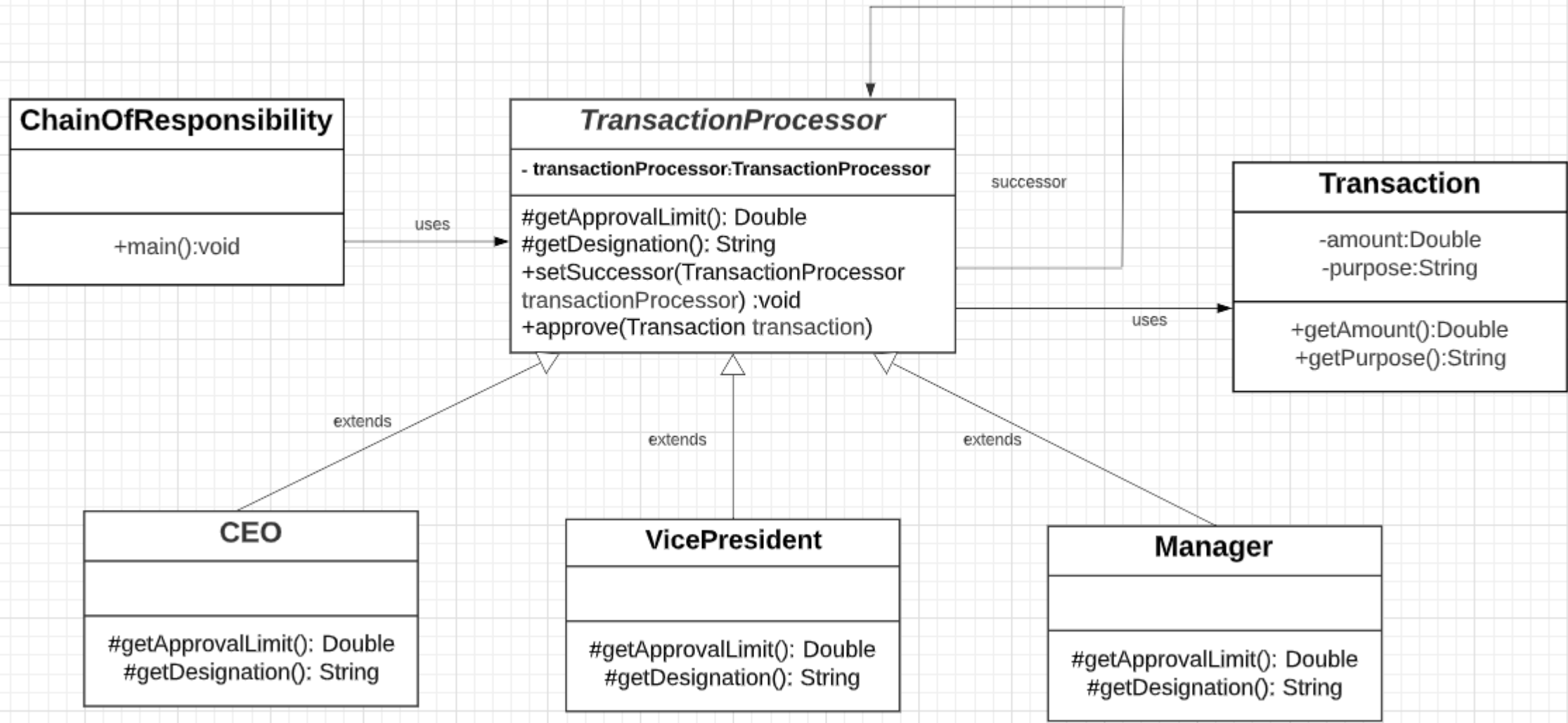
All transactions lesser than 1,00,000 amount can be approved by manager, lesser than 10,00,000 can be approved by vice president and lesser than 25,00,000 can be approved by CEO.

The transaction approval behavior can be implemented with a simple if else conditions, by checking if amount is less than 1,00,000 or else if amount is less than 10,00,000 and so on. But this approach is static, means we can not change these conditions dynamically. Chain of responsibility design pattern can be used in this situation.

Chain of Responsibility

We need to design processing and command objects. Based on the above example, processing objects are employees like Manager, Vice President because they process the transaction by approving and command object is the transaction. Once processing objects are created then we need to chain them together like **manager** -> **vice president** -> **CEO** and pass the transaction at the beginning of the chain. Transaction continues to flow in the chain until it reaches the employee, who's limit allows to approve it.

Chain of Responsibility



Proxy versus Chain of Responsibility

- Proxy represents a single object while Chain of Responsibility can have many objects.
- Proxy first received the request from the client but never processed by the proxy object, while in Chain of Responsibility it processes by the target object. In the Chain of Responsibility, an object that received the client request could process the request first.
- Proxy always forward client request to target object while in CoR forward the request in the chain only current object can not process the request.
- In Proxy response to the request guaranteed provided communication link between client and server location working while in CoR Response to request is not guaranteed. It means the request might reach the end of the chain but the request could not be processed.

The Command Pattern

The definition of Command provided in the original Gang of Four book on DesignPatterns states:

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

The Command Pattern

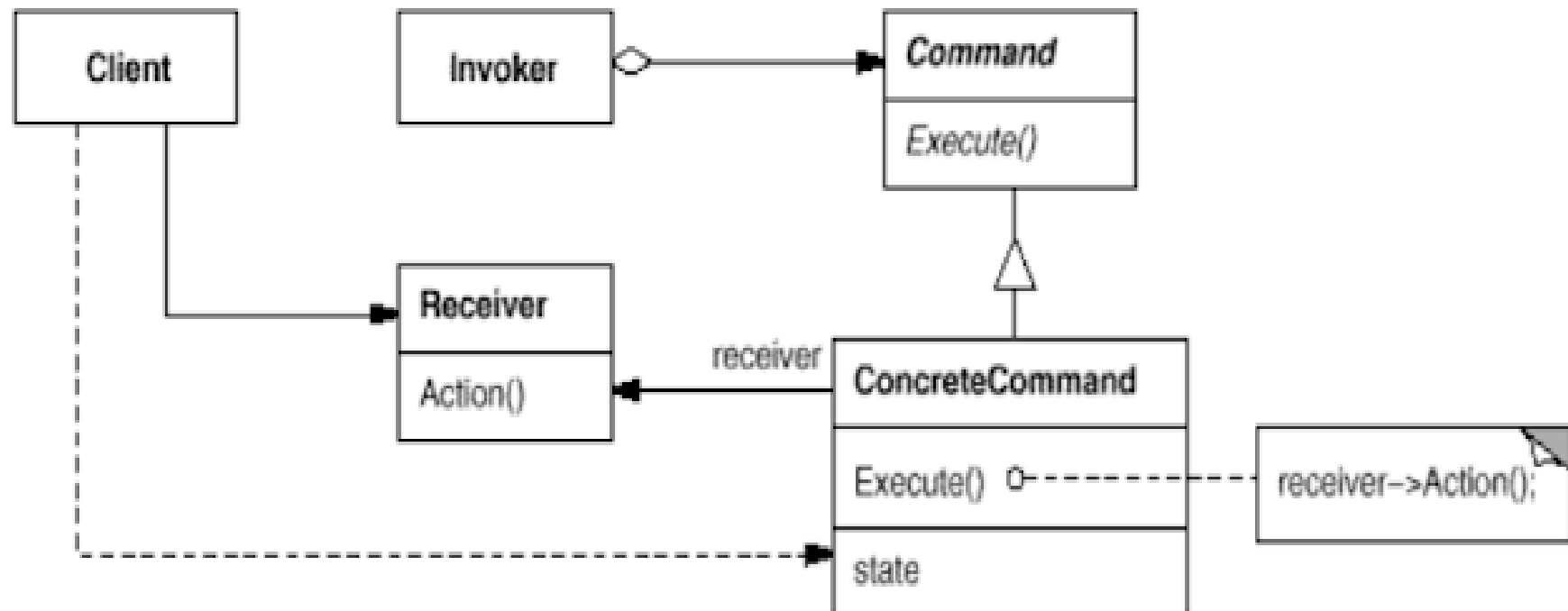
- The pattern intends to encapsulate in an object all the data required for performing a given action (command), including what method to call, the method's arguments, and the object to which the method belongs.
- This model allows us to decouple objects that produce the commands from their consumers, so that's why the pattern is commonly known as the producer-consumer pattern.
- The similar approach is adapted into chain of responsibility. Only difference is that in command there is one request handler, and in chain of responsibility there can be many handlers for single request object.

The Command Pattern

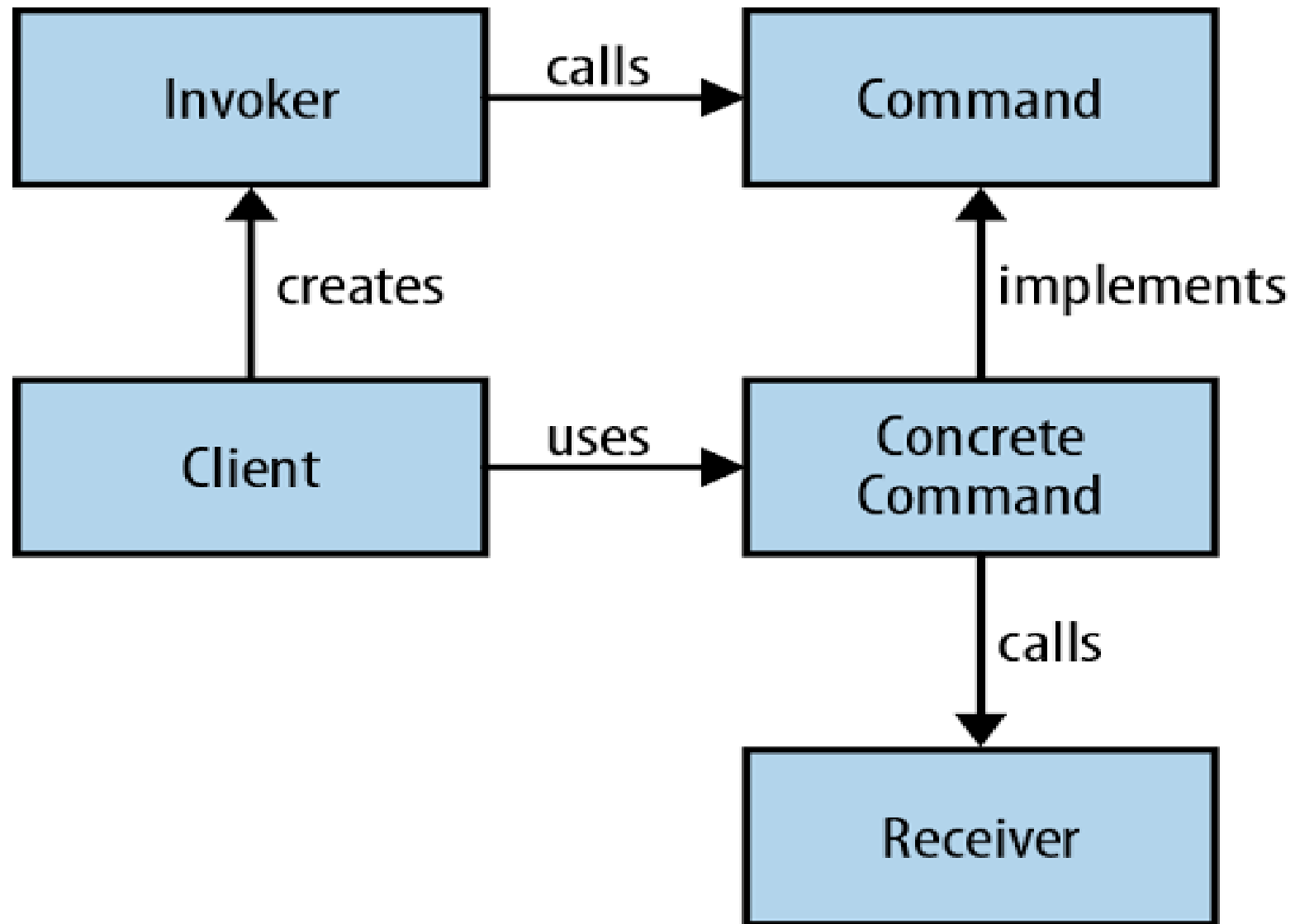
The command pattern requires implementing the following components:

the Command, the Receiver, Concrete Command, the Invoker, and the Client

Participants



The Command Pattern

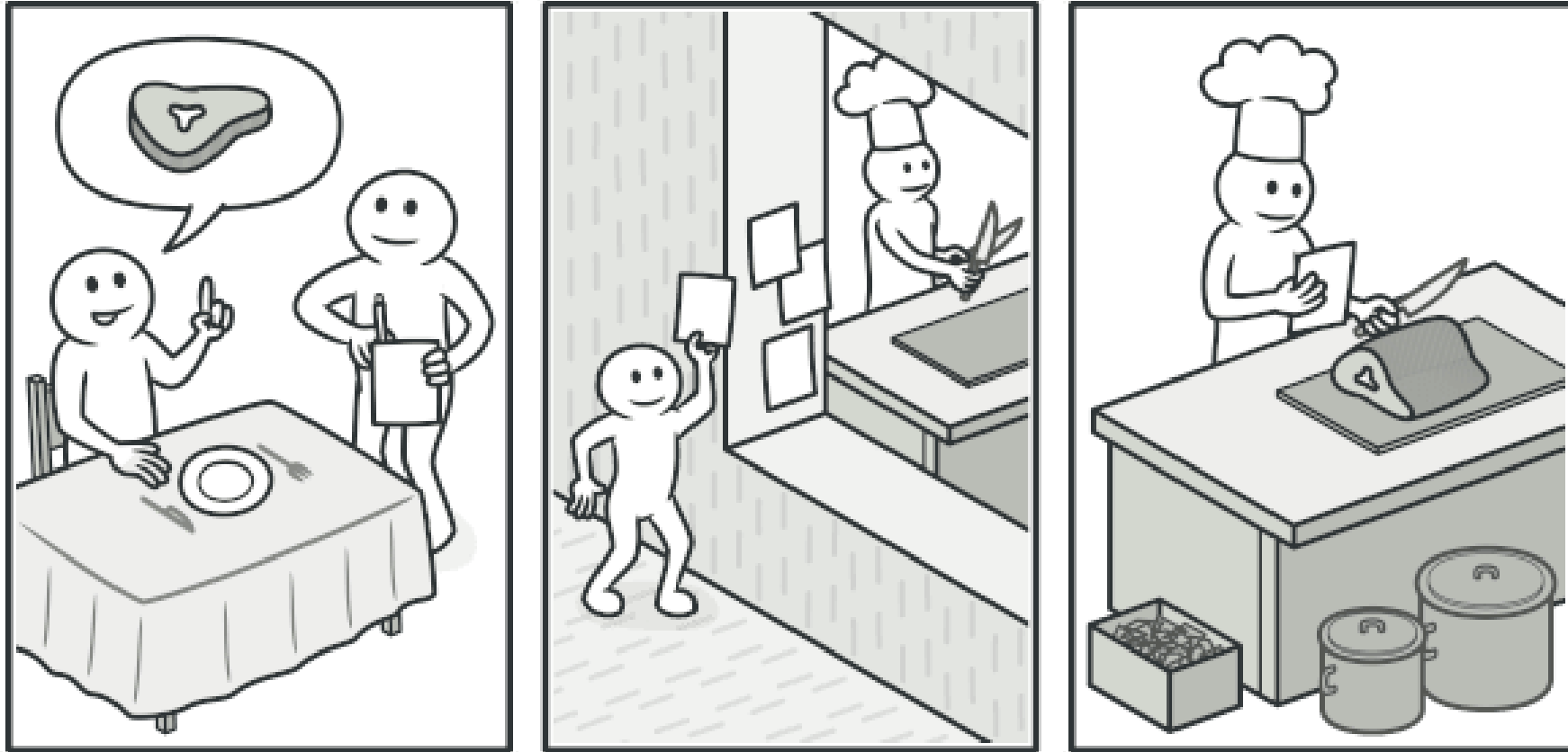


The Command Pattern

Collaborations

- The client creates a *ConcreteCommand* object and specifies its receiver.
- An Invoker object stores the *ConcreteCommand* object.
- The invoker issues a request by calling *Execute* on the command.
When commands are undoable, *ConcreteCommand* stores state for undoing the command prior to invoking *Execute*.
- The *ConcreteCommand* object invokes operations on its receiver to carry out the request.

The Command Pattern



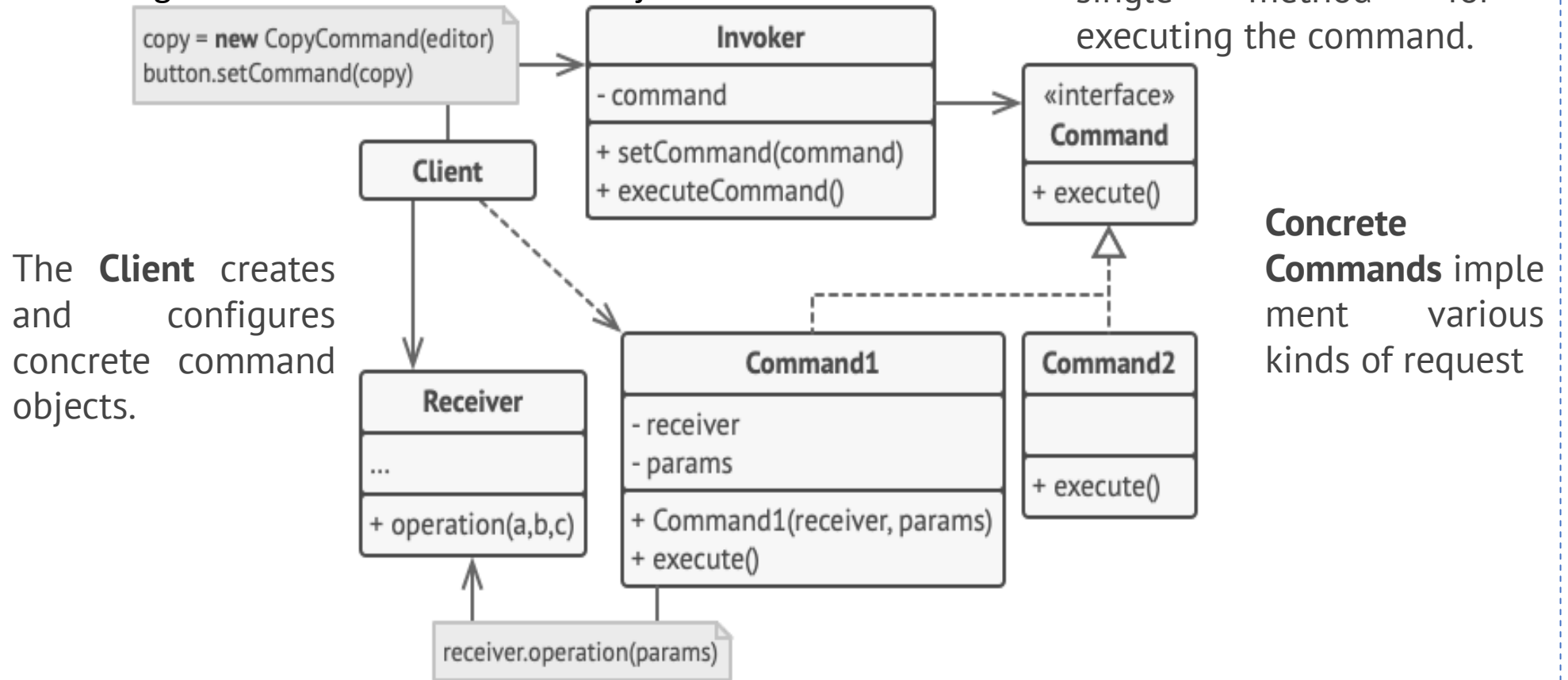
The Command Pattern

- Consider the scenario when a customer goes to a restaurant and wants to place an order for a meal. The waiter merely writes the order he gets on a piece of paper and passes it on to the chef. The chef executes the order and then prepares the meal. He passes the piece of paper to the manager.
- The verbal order from the customer has now become a paper object. This piece of paper is the command object. The command object contains all the details needed to execute the request.
- In object-oriented programming, we can encapsulate all the details of a request into an object and pass that object to execute it.

The Command Pattern

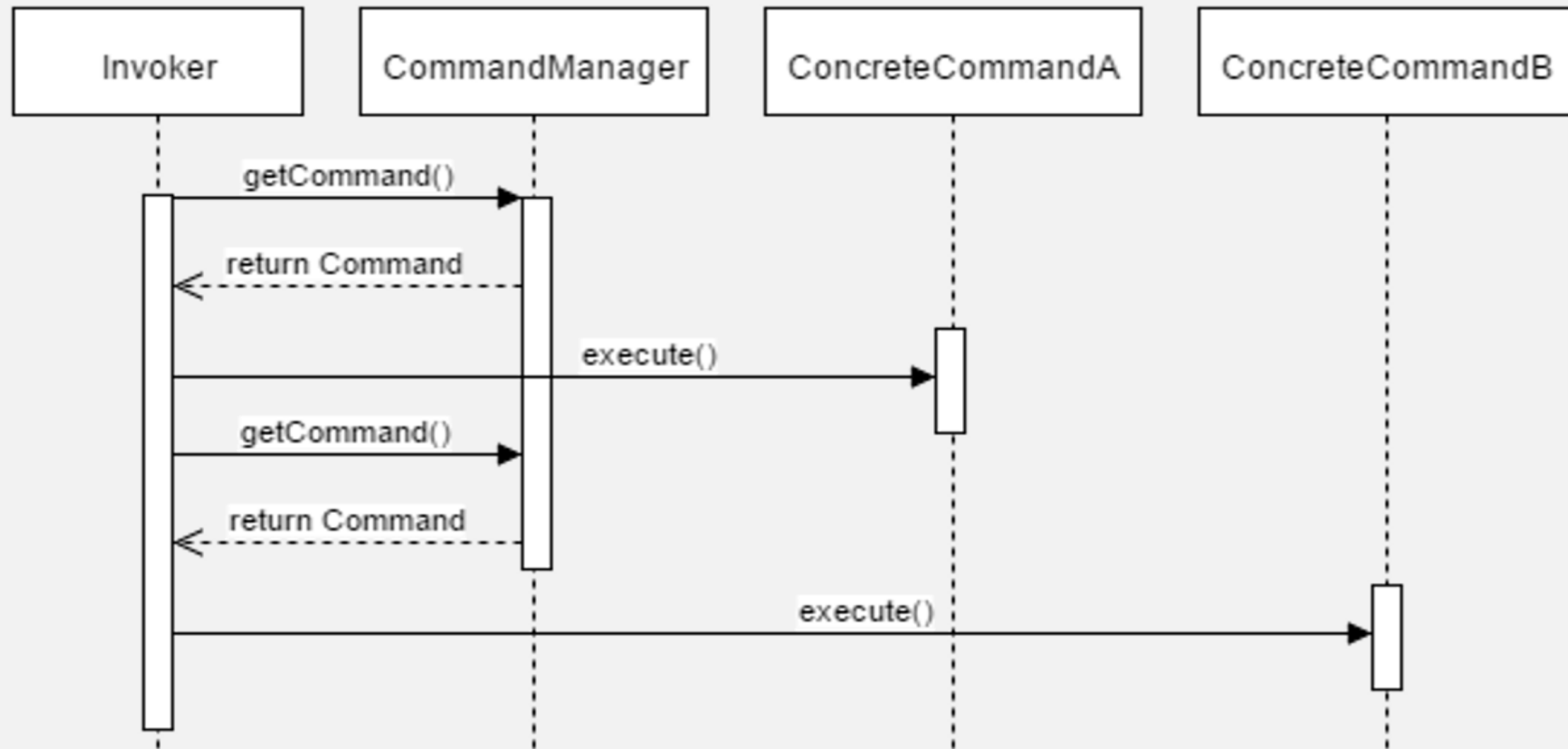
The **Sender** class (aka invoker) is responsible for initiating requests. This class must have a field for storing a reference to a command object.

The **Command** interface usually declares just a single method for executing the command.



The **Receiver** class contains some business logic. Almost any object may act as a receiver

Command pattern – Diagram of sequence



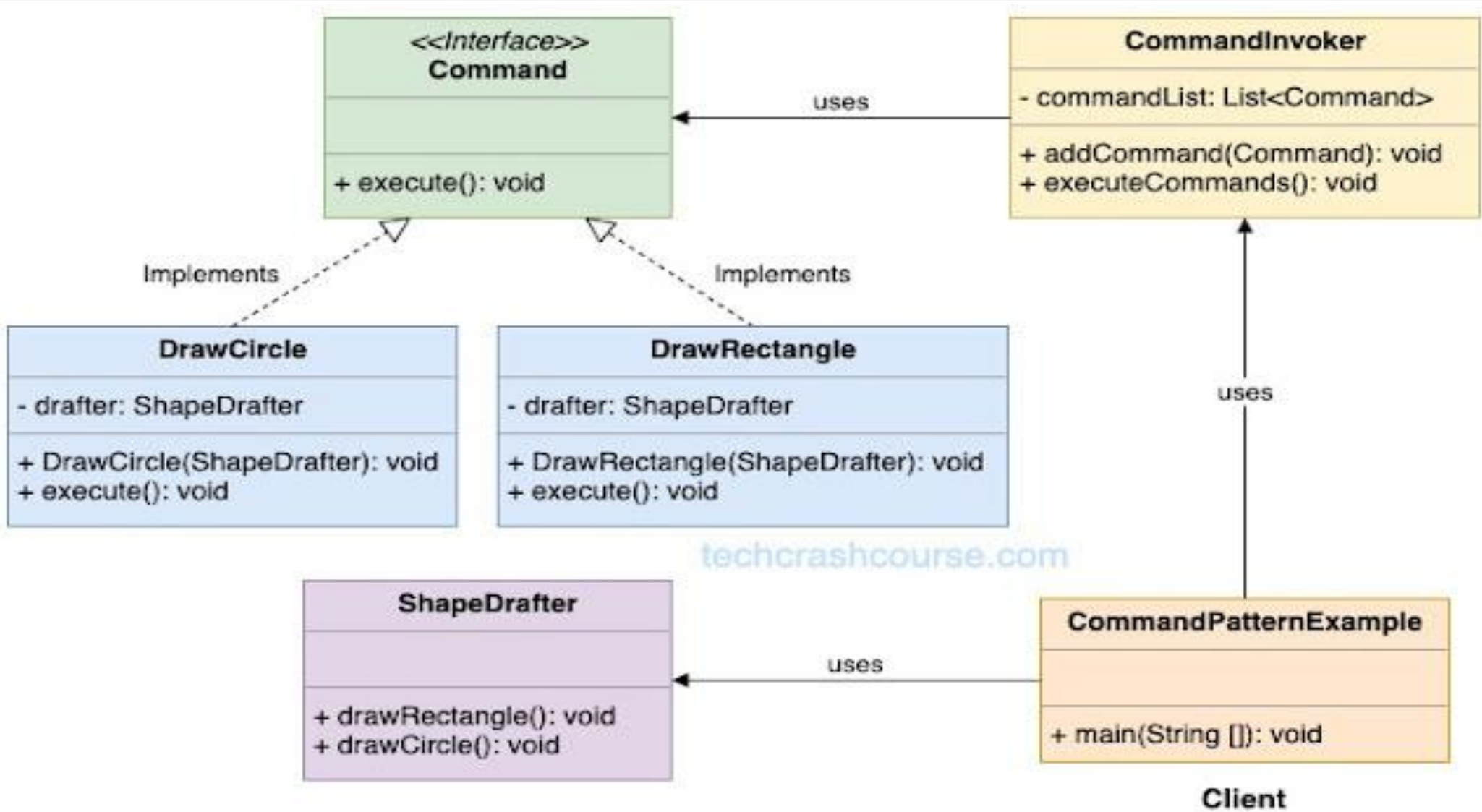
Advantages of Command Pattern

- A command decouples the command invoker object and the one who actually perform the command. The invoker invokes the command without knowing about details of who or how it is going to be executed.
- For a invoker, all command objects are same. It just call the execute method of every command.
- We can add a new command any time without any modification in invoker or client implementation.

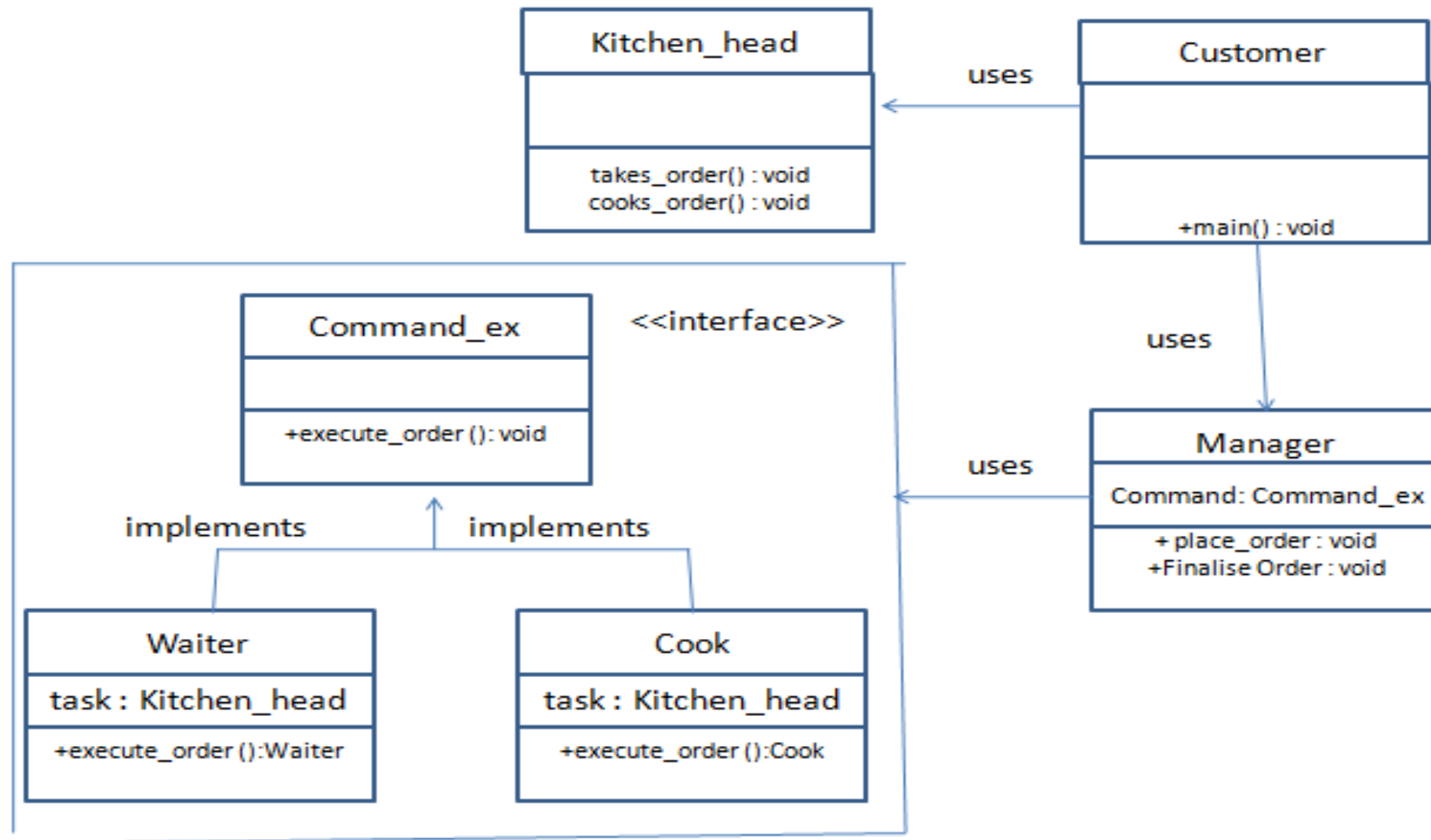
When we should use Command Pattern

- When we want to keep historical records of the actions performed on an object, where every action is implemented as a command.
- When we implement a sequence of tasks as workflow. We can model every task as a command and execute them in a specific sequence.
- When we want to implement object oriented call back.
- When we want to decouple the invoker and the receiver.

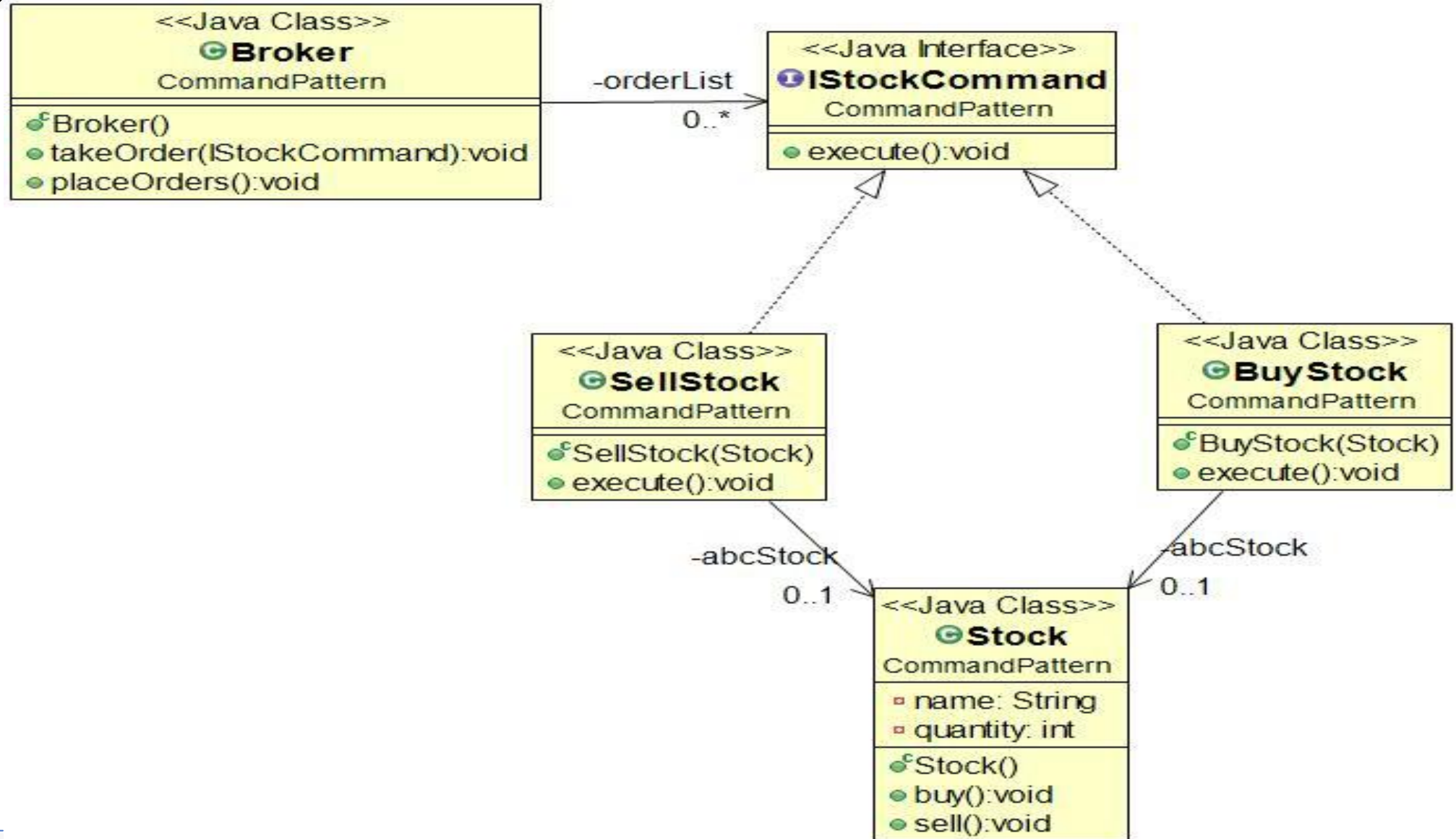
The Command Pattern

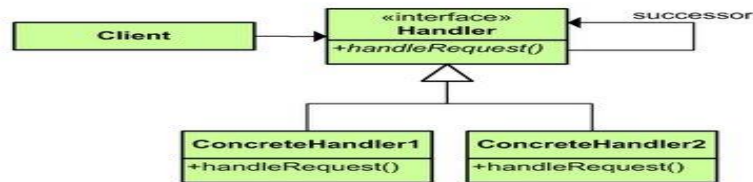


The Command Pattern



The Command Pattern

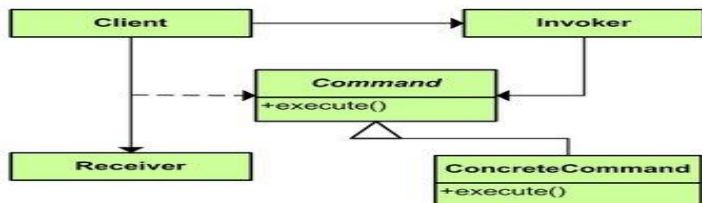




Chain of Responsibility

Type: Behavioral

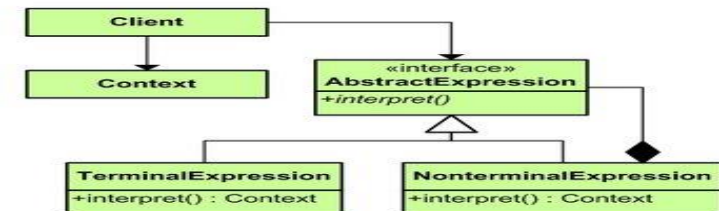
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Command

Type: Behavioral

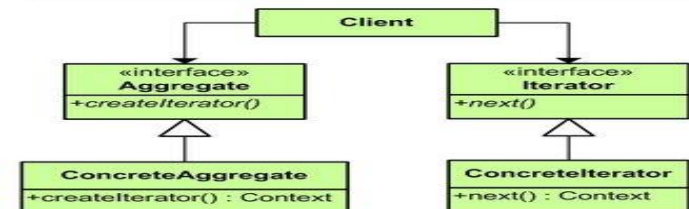
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



Interpreter

Type: Behavioral

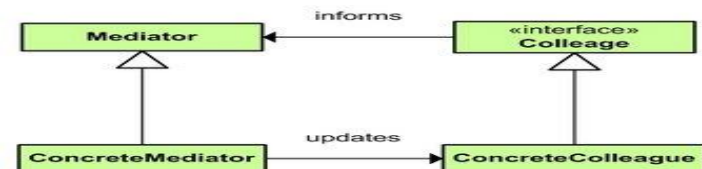
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Iterator

Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Mediator

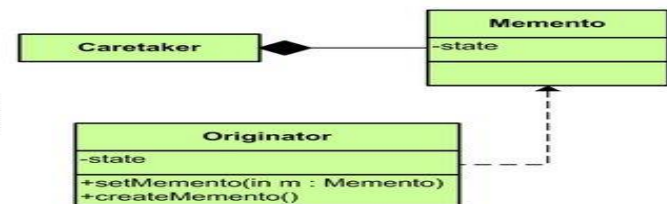
Type: Behavioral

What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.

Memento

Type: Behavioral

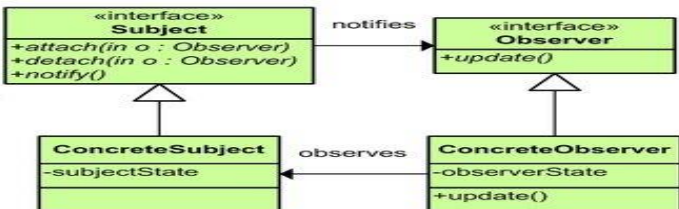
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Observer

Type: Behavioral

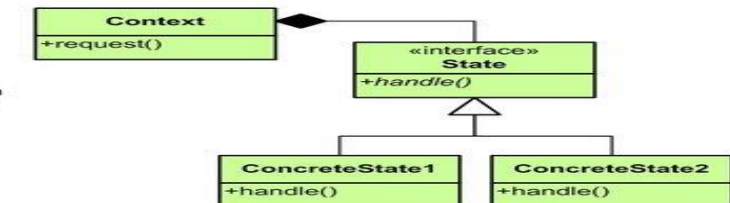
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



State

Type: Behavioral

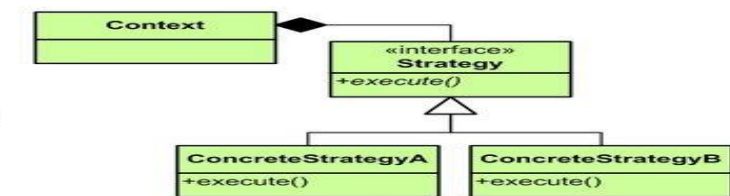
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Strategy

Type: Behavioral

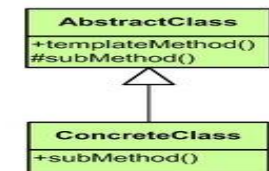
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Template Method

Type: Behavioral

What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

