Unidade Curricular

“Padrões e Desenho de Software”

#11 – Behavioral Patterns (1)

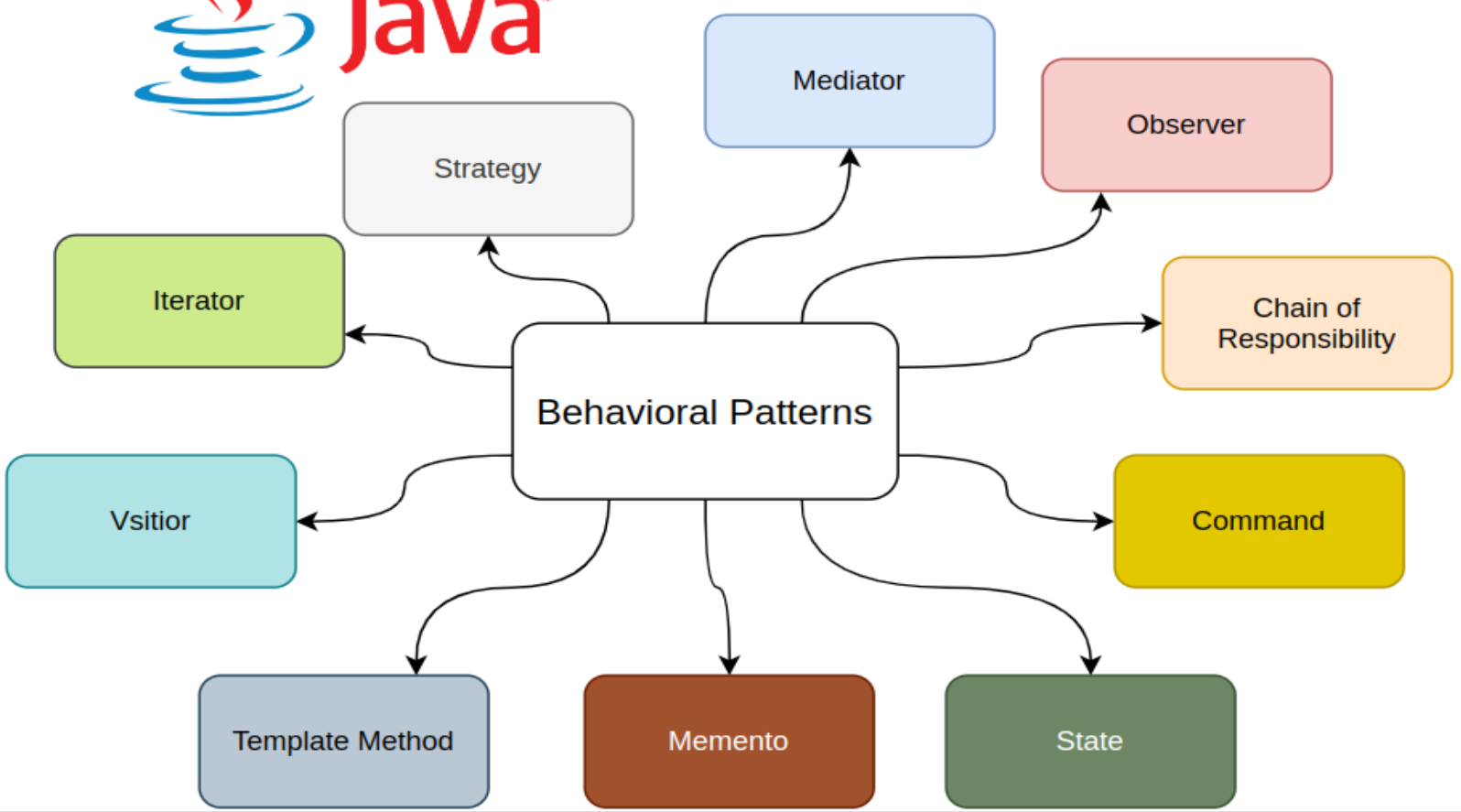
António José Ribeiro Neves

an@ua.pt

https://www.ua.pt/pt/uc/12275

Outline

Behavioral Patterns - overview



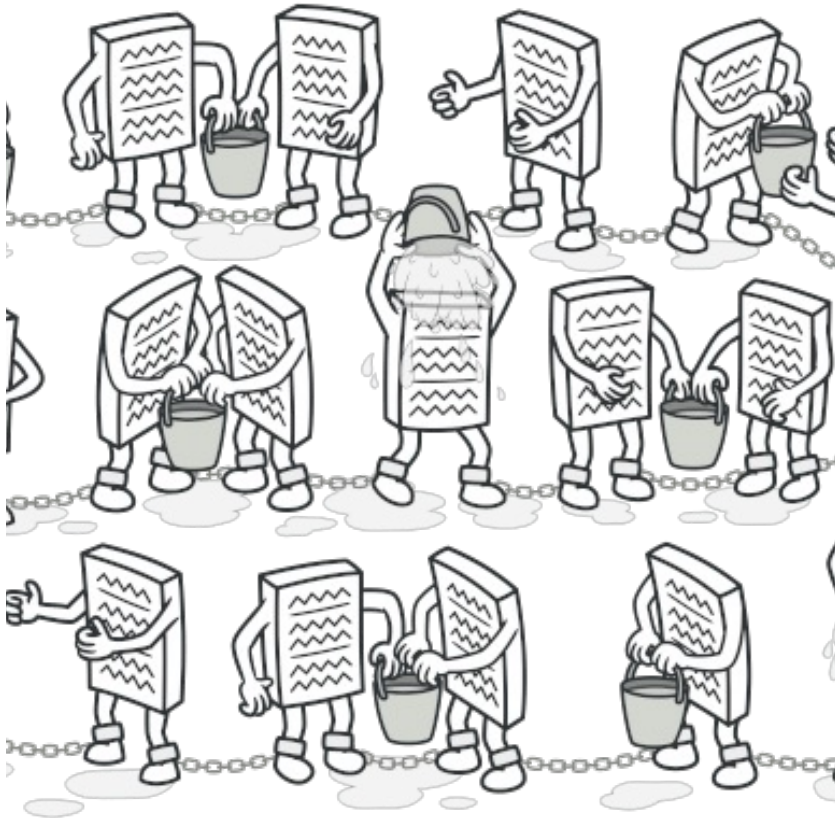
Behavioral Design Patterns

- **100 minutes** to explore the Adapter Design Pattern and answer the following questions for each pattern:

https://forms.gle/xtMWCg5iYP3deXt78

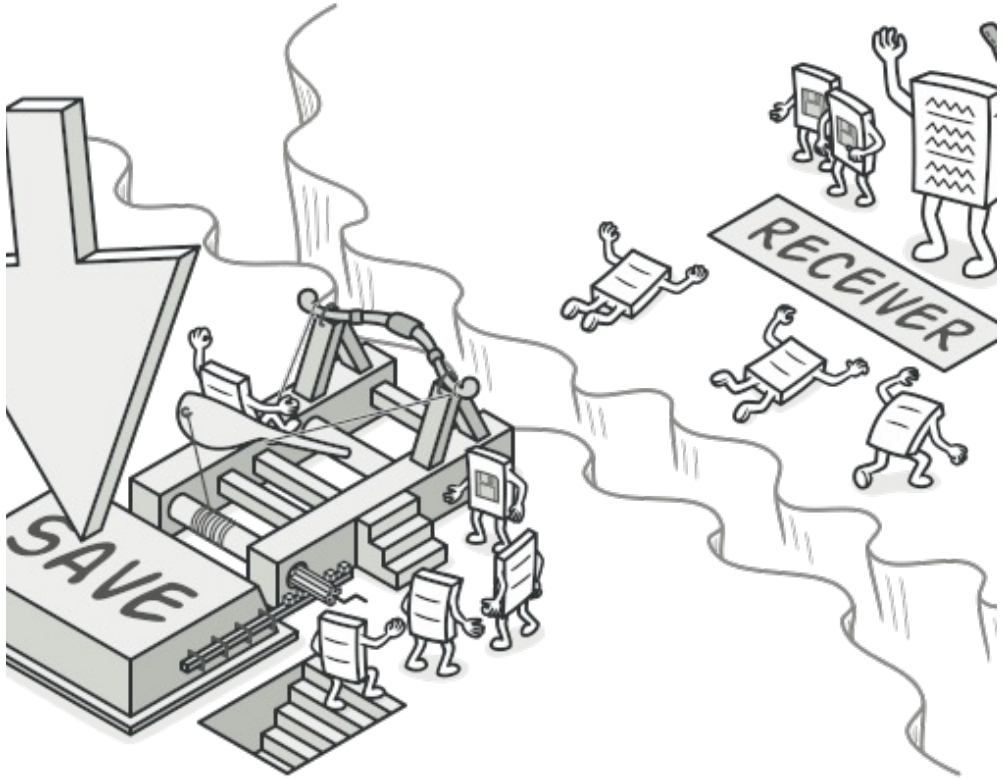
Chain of Responsibility

- Allows an object to pass a request along a chain of handlers.
- Handlers can either handle the request or pass it to the next handler in the chain.
- Decouples senders and receivers of a request.
- Allows for dynamic addition or removal of handlers without affecting the client.
- **Key Components:**
  - **Handler Interface**: Defines the interface for handling requests and optionally chaining to the next handler.
  - **Concrete Handlers**: Implement the Handler interface and handle requests. They may also pass requests to the next handler.
  - **Client**: Initiates the request and starts the chain of handlers.
- **How it Works:**
  - Client sends a request to the first handler in the chain.
  - Each handler decides whether to handle the request or pass it to the next handler.
  - The request is passed along the chain until it is handled or the end of the chain is reached.



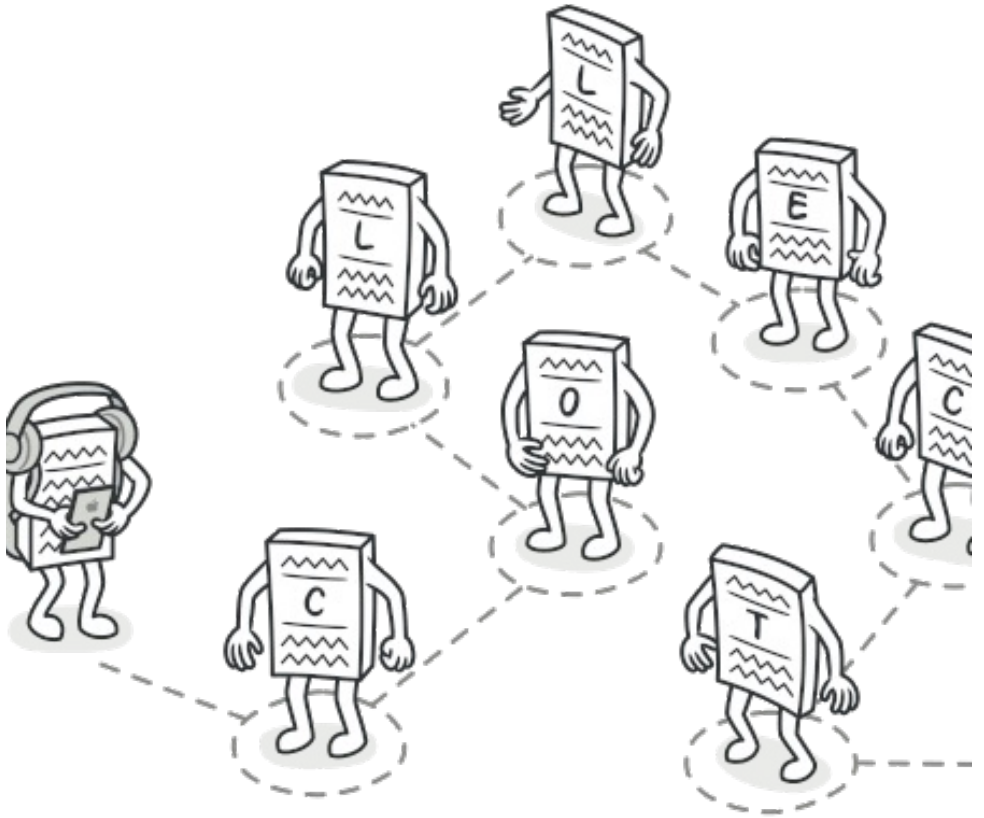
Command

- Encapsulates a request as an object, thereby allowing parameterization of clients with queues, requests, and operations.
- Decouples the sender of a request from the receiver, allowing for parameterization of clients with different requests.
- **Key Components:**
  - **Command**: Defines an interface for executing an operation.
  - **Concrete Command**: Implements the Command interface and encapsulates the receiver (object) and the action to be performed.
  - **Invoker**: Asks the command to carry out the request.
  - **Receiver**: Knows how to perform the operations associated with a request.
- **How it Works:**
  - Client creates a command object and associates it with a receiver.
  - Invoker is responsible for sending the command to the receiver.
  - Receiver executes the command.



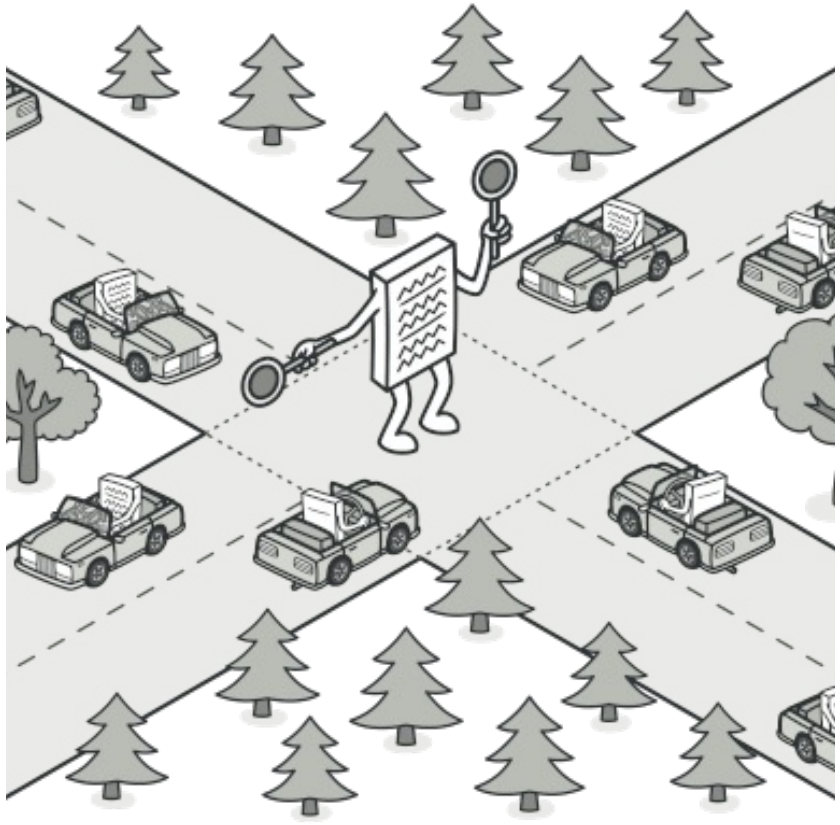
Iterator

- Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Allows for traversal of a collection of objects without needing to know its internal structure.
- **Key Components:**
  - **Iterator**: Defines an interface for accessing and traversing elements.
  - **Concrete Iterator**: Implements the Iterator interface and keeps track of the current position in the traversal.
  - **Aggregate**: Defines an interface for creating an Iterator object.
  - **Concrete Aggregate**: Implements the Aggregate interface and provides the mechanism for creating Iterator objects.
- Client requests an Iterator object from the Aggregate.
- Iterator traverses the elements of the collection sequentially.
- Client uses the Iterator to access the elements of the collection without knowledge of its internal structure.



Mediator

- Defines an object that encapsulates how a set of objects interact, promoting loose coupling by keeping objects from referring to each other explicitly.
- Centralizes complex communication and control logic between multiple objects into a single mediator object.
- **Key Components:**
  - **Mediator**: Defines an interface for communicating with Colleague objects.
  - **Concrete Mediator**: Implements the Mediator interface and coordinates communication between Colleague objects.
  - **Colleague**: Defines an interface for interacting with other Colleague objects.
  - **Concrete Colleague**: Implements the Colleague interface and communicates with other Colleague objects through the Mediator.
- **How it Works:**
  - Colleague objects communicate with each other indirectly through the Mediator.
  - When a Colleague needs to communicate with another Colleague, it sends a message to the Mediator.
  - The Mediator handles the message and forwards it to the appropriate Colleague(s).

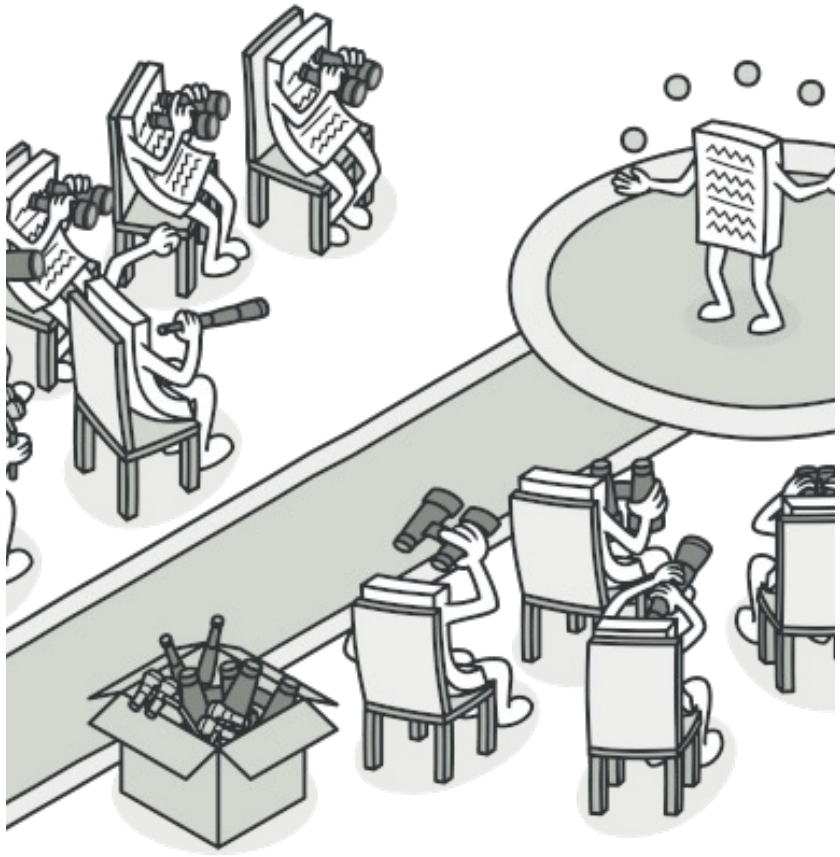


Memento

- Captures and externalizes an object's internal state without violating encapsulation, allowing the object to be restored to its previous state.
- Supports undo/redo functionality by storing and managing multiple states.
- Simplifies the implementation of complex state management logic.
- **Key Components:**
  - **Originator**: The object whose state needs to be saved and restored.
  - **Memento**: Stores the state of the Originator object.
  - **Caretaker**: Manages the Memento objects, but does not modify or inspect their contents.
- **How it Works:**
  - Originator creates a Memento object to store its state.
  - Originator can use Memento to restore its state to a previous state.
  - Caretaker is responsible for managing the Memento objects, typically storing them in a stack or list.

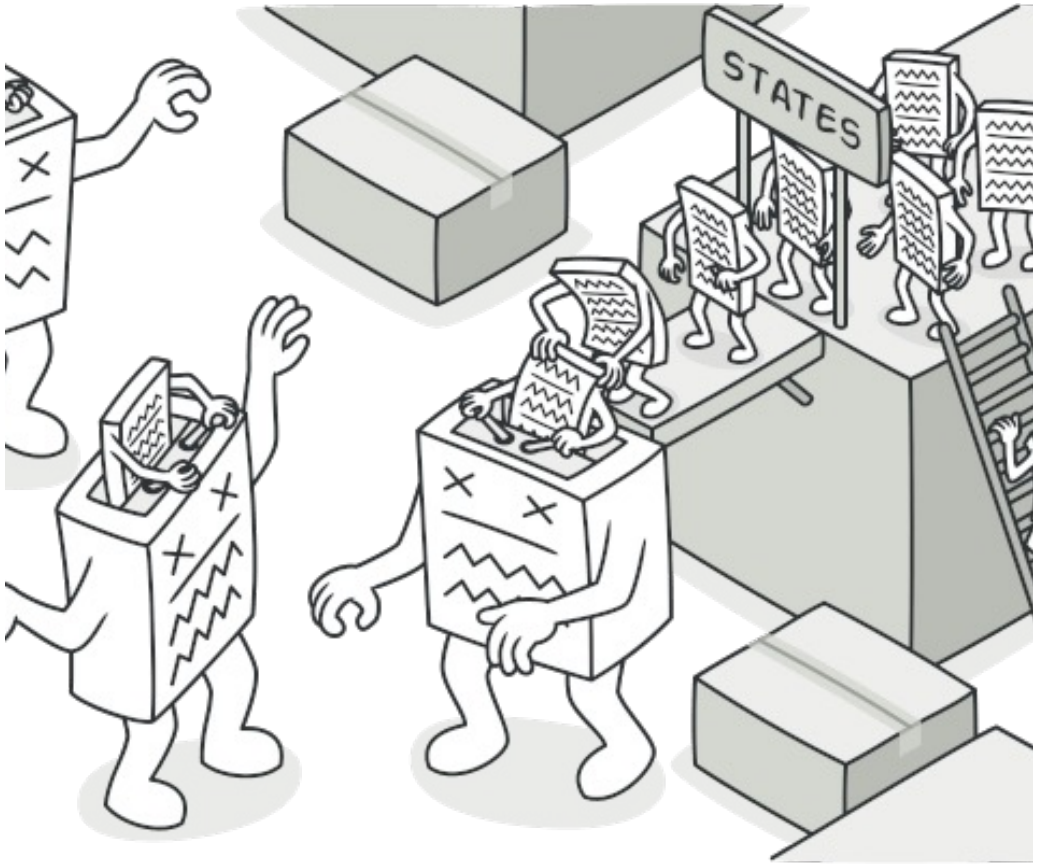
Observer

- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Key Components:**
  - **Subject (Observable)**: Maintains a list of observers and notifies them of state changes.
  - **Observer**: Defines an interface for receiving notifications from the Subject.
  - **Concrete Subject**: Implements the Subject interface and manages the state being observed.
  - **Concrete Observer**: Implements the Observer interface and receives notifications from the Subject.
- **How it Works:**
  - Observers register themselves with the Subject to receive notifications.
  - When the state of the Subject changes, it notifies all registered Observers.
  - Observers update their state or perform actions based on the notification received from the Subject.



State

- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Key Components:**
  - **Context**: Represents the object whose behavior changes based on its internal state.
  - **State**: Defines an interface for encapsulating the behavior associated with a particular state of the Context.
  - **Concrete State**: Implements the State interface and defines the behavior associated with a specific state.
- **How it Works:**
  - Context delegates state-specific behavior to a State object.
  - Context may change its current state object if its internal state changes.
  - Each Concrete State object encapsulates behavior for a specific state of the Context.

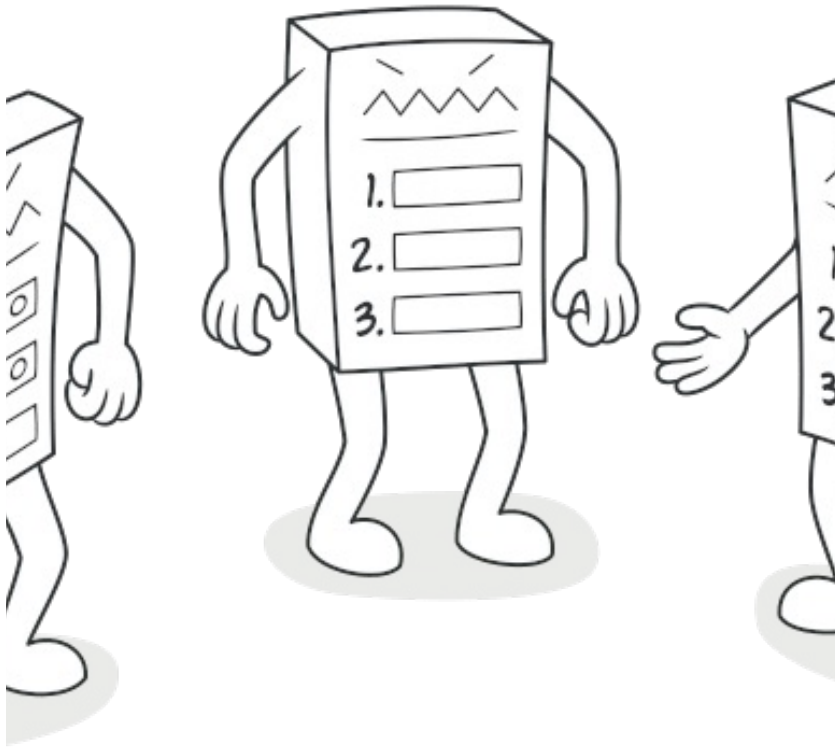


Strategy

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable. The strategy pattern lets the algorithm vary independently from clients that use it.
- **Key Components:**
  - **Context**: Represents the client that uses a Strategy object.
  - **Strategy**: Defines an interface for a family of algorithms.
  - **Concrete Strategy**: Implements the Strategy interface and provides a specific algorithm.
- **How it Works:**
  - Context maintains a reference to a Strategy object.
  - Context delegates the execution of a particular algorithm to the Strategy object.
  - The Strategy object encapsulates the algorithm's implementation details.

Template Method

- Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of an algorithm without changing its structure.
- **Key Components:**
  - **Abstract Class (Template)**: Defines the skeleton of the algorithm with template methods.
  - **Concrete Class**: Implements the abstract class and overrides specific steps of the algorithm.
- **How it Works:**
  - The abstract class contains a template method that defines the algorithm's structure.
  - Some steps of the algorithm are implemented directly in the abstract class, while others are declared as abstract methods to be implemented by subclasses.
  - Concrete subclasses override the abstract methods to provide specific implementations for certain steps, while reusing the common structure defined by the template method.



Visitor

- Allows for the separation of an algorithm from the objects on which it operates. It enables adding new operations to existing object structures without modifying them.
- **Key Components:**
  - **Visitor**: Defines the interface for visiting each element in the object structure.
  - **Concrete Visitor**: Implements the Visitor interface and defines the operations to be performed on each element.
  - **Element**: Defines the interface for accepting visitors.
  - **Concrete Element**: Implements the Element interface and provides an implementation for accepting visitors.
  - **Object Structure**: Represents a collection of elements to be visited.
- **How it Works:**
  - The Object Structure contains a collection of elements.
  - Each element exposes an accept method that takes a visitor as an argument.
  - The visitor interface defines visit methods for each type of element.
  - Concrete visitors implement the visit methods to perform specific operations on elements.