4

Behavioral Design Patterns

Objectives

After completing this lesson, you should be able to do the following:

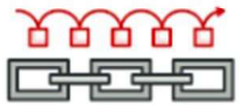
- Analysis of Behavioral DP
- Exploration of Chain of Responsibility Pattern, Command Pattern
- Iterator Pattern, Mediator Pattern
- State Pattern, Strategy Pattern, Template Pattern
- Memento Pattern, Visitor Pattern, Observer Pattern

Course Roadmap

Java Design Patterns

- Lesson 1: Introduction to Design Patterns
- Lesson02: Creational Design Patterns
- Lesson03: Structural Design Patterns
- **Lesson04: Behavioral Design Patterns** — **You are here!**
- Lesson 5: Most Useful Design Patterns

Behavioral Design Patterns



Chain of Responsibility

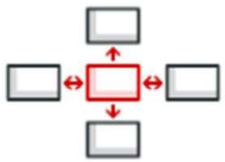
Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

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.

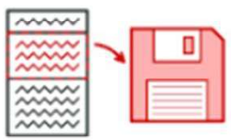
Iterator

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



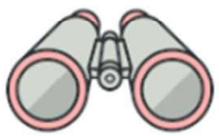
Mediator

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.



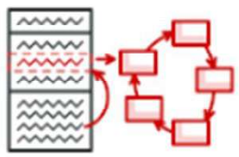
Memento

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



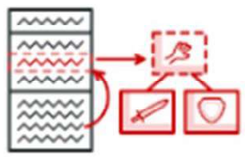
Observer

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



State

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

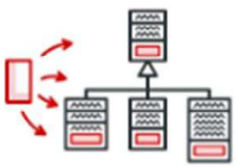


Strategy

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.

Introduction

- ***Behavioral design patterns*** *are concerned with algorithms and the assignment of responsibilities between objects.*

- *Behavioral patterns provide a solution for better interaction between objects and how to provide loose-coupling and flexibility to extend easily.*

Chain of Responsibility Pattern

Introduction

- ***Chain of Responsibility*** *is behavioral design pattern that allows passing request along the chain of potential handlers until one of them handles request.*

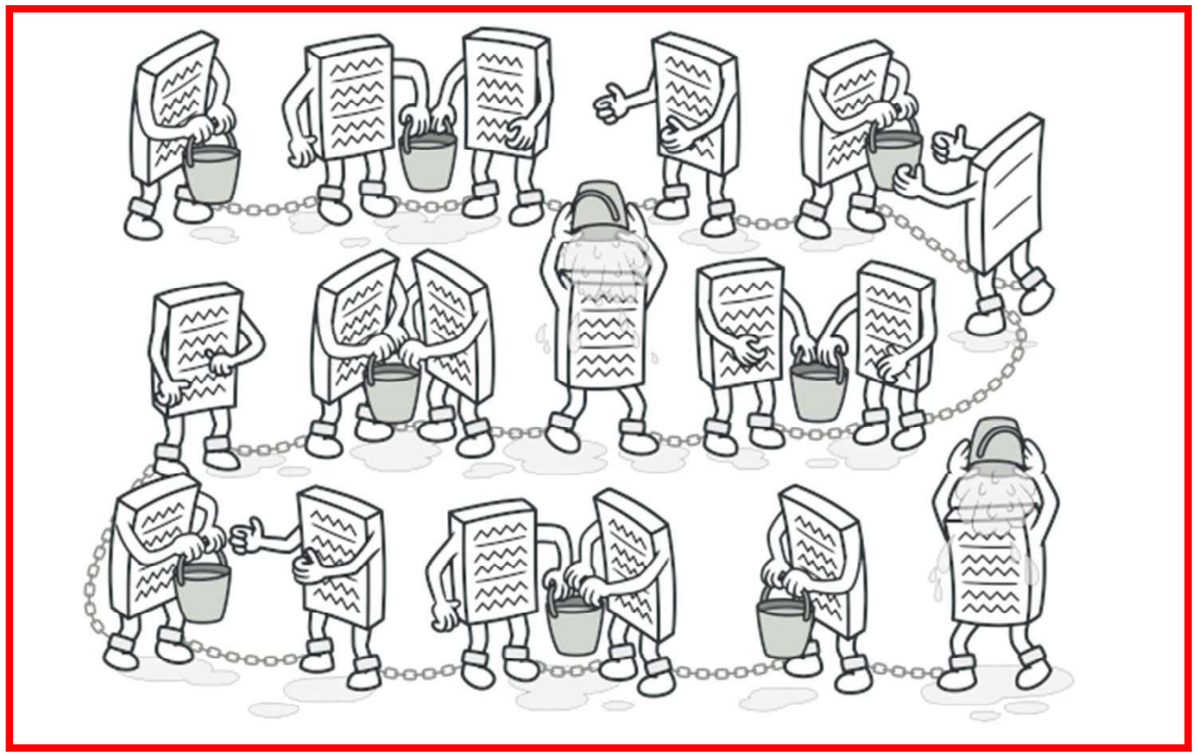
- *The chain of responsibility pattern is used to achieve loose-coupling in software design where a request from the client is passed to a chain of objects to process them. Then the object in the chain will decide who will be processing the request and whether the request is required to be sent to the next object in the chain or not.*

Illustration

- We know that we can have multiple catch blocks in a try-catch block code. Here every catch block is kind of a processor to process that particular exception. So when an exception occurs in the try block, it's sent to the first catch block to process.
- If the catch block is not able to process it, it forwards the request to the next Object in the chain (i.e., the next catch block). If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program.

Intent

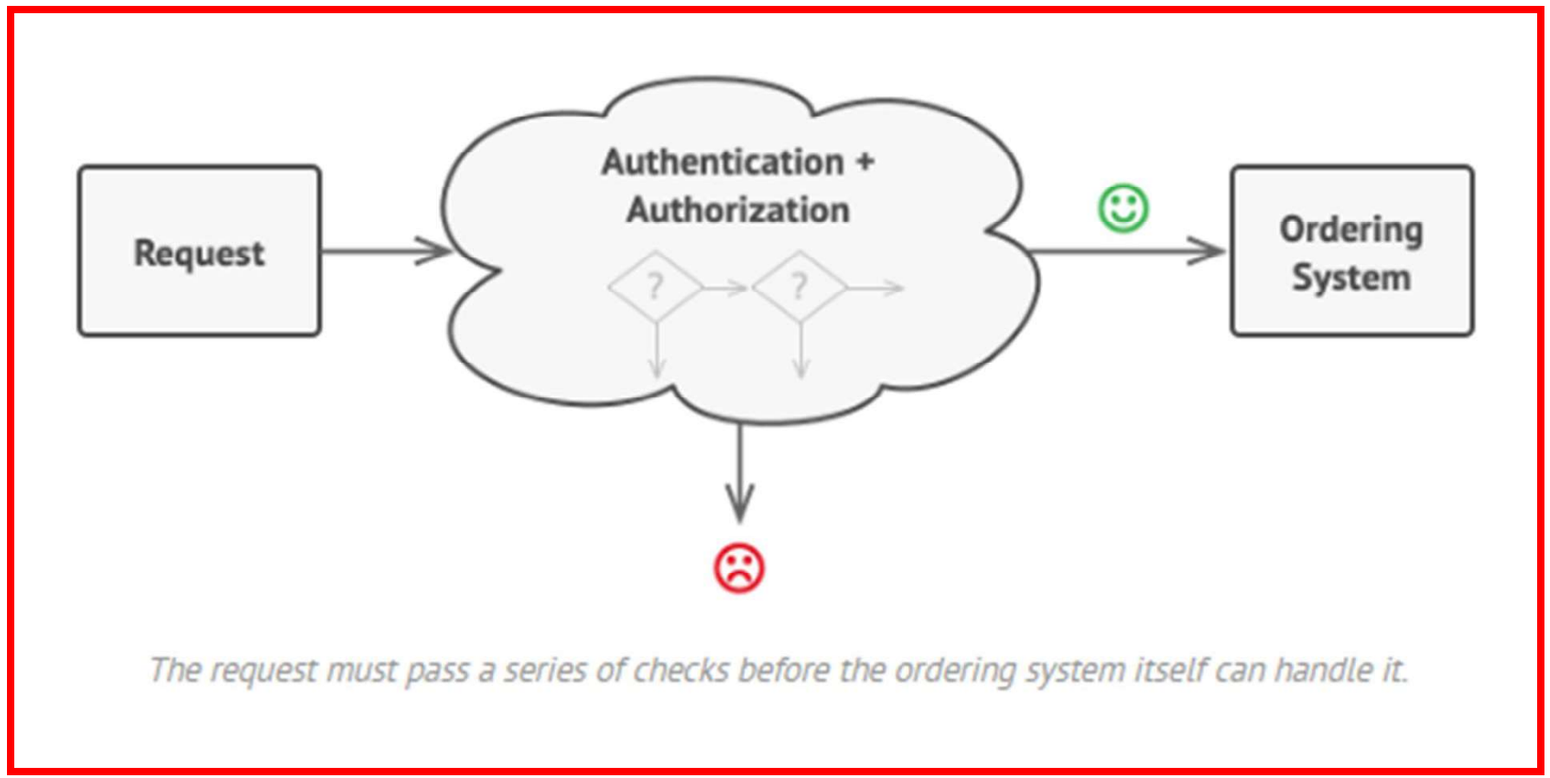
- **Chain of Responsibility** is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.



Problem

- Imagine that you're working on an online ordering system. You want to restrict access to the system so only authenticated users can create orders. Also, users who have administrative permissions must have full access to all orders.

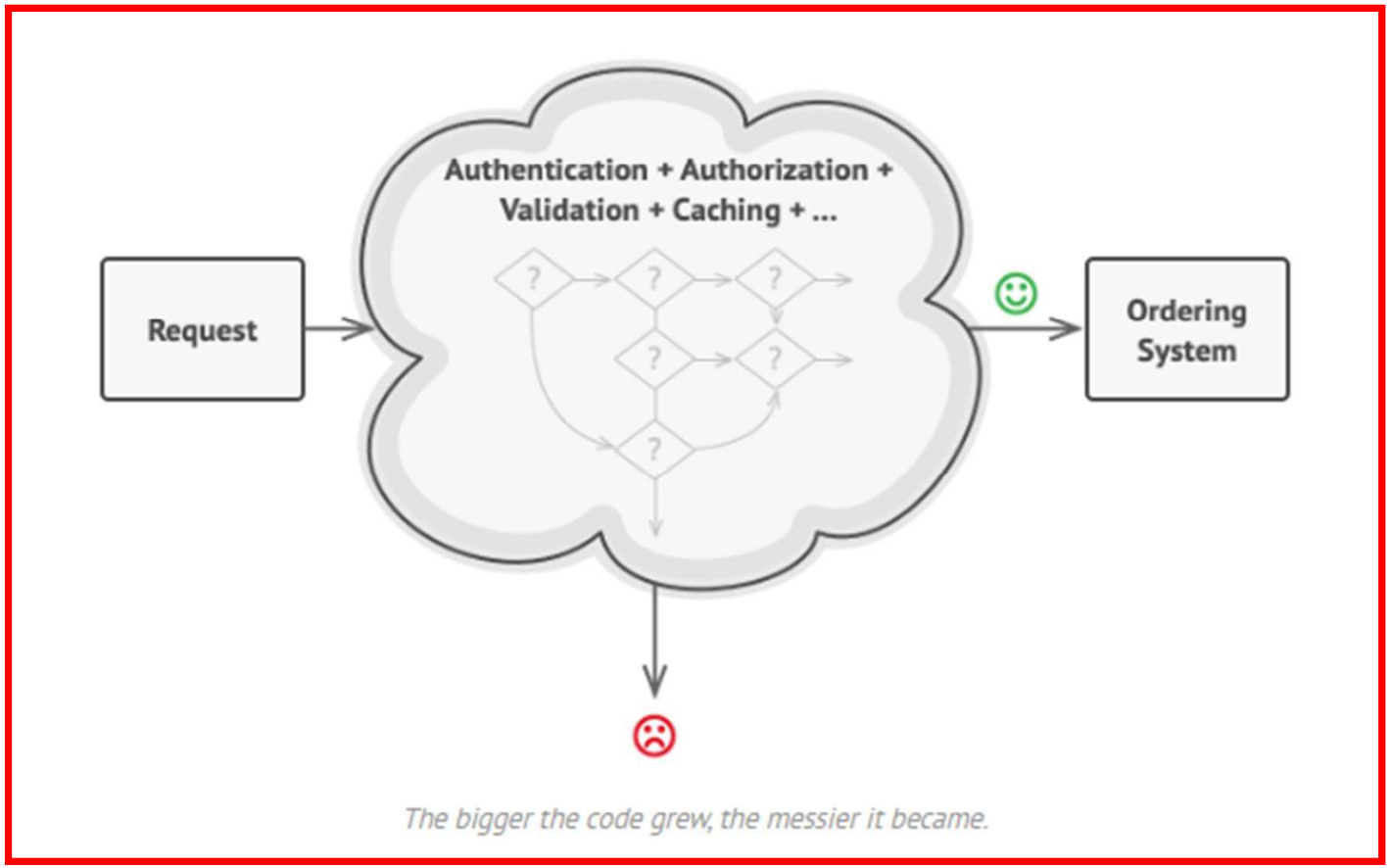
- After a bit of planning, you realized that these checks must be performed sequentially. The application can attempt to authenticate a user to the system whenever it receives a request that contains the user's credentials. However, if those credentials aren't correct and authentication fails, there's no reason to proceed with any other checks.



The request must pass a series of checks before the ordering system itself can handle it.

During the next few months, you implemented several more of those sequential checks.

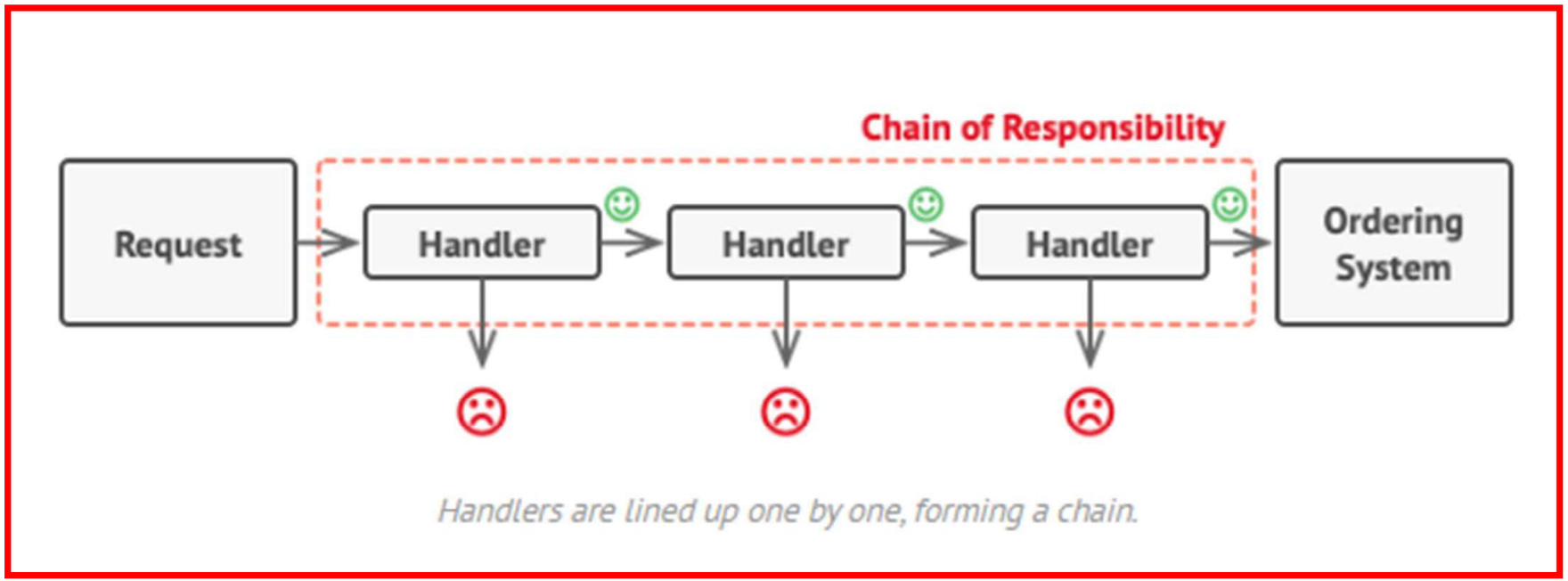
- One of your colleagues suggested that it's unsafe to pass raw data straight to the ordering system. So you added an extra validation step to sanitize the data in a request.
- Later, somebody noticed that the system is vulnerable to brute force password cracking. To negate this, you promptly added a check that filters repeated failed requests coming from the same IP address.
- Someone else suggested that you could speed up the system by returning cached results on repeated requests containing the same data. Hence, you added another check which lets the request pass through to the system only if there's no suitable cached response.


Authentication + Authorization +
Validation + Caching + ...
Request
Ordering System
The bigger the code grew, the messier it became.

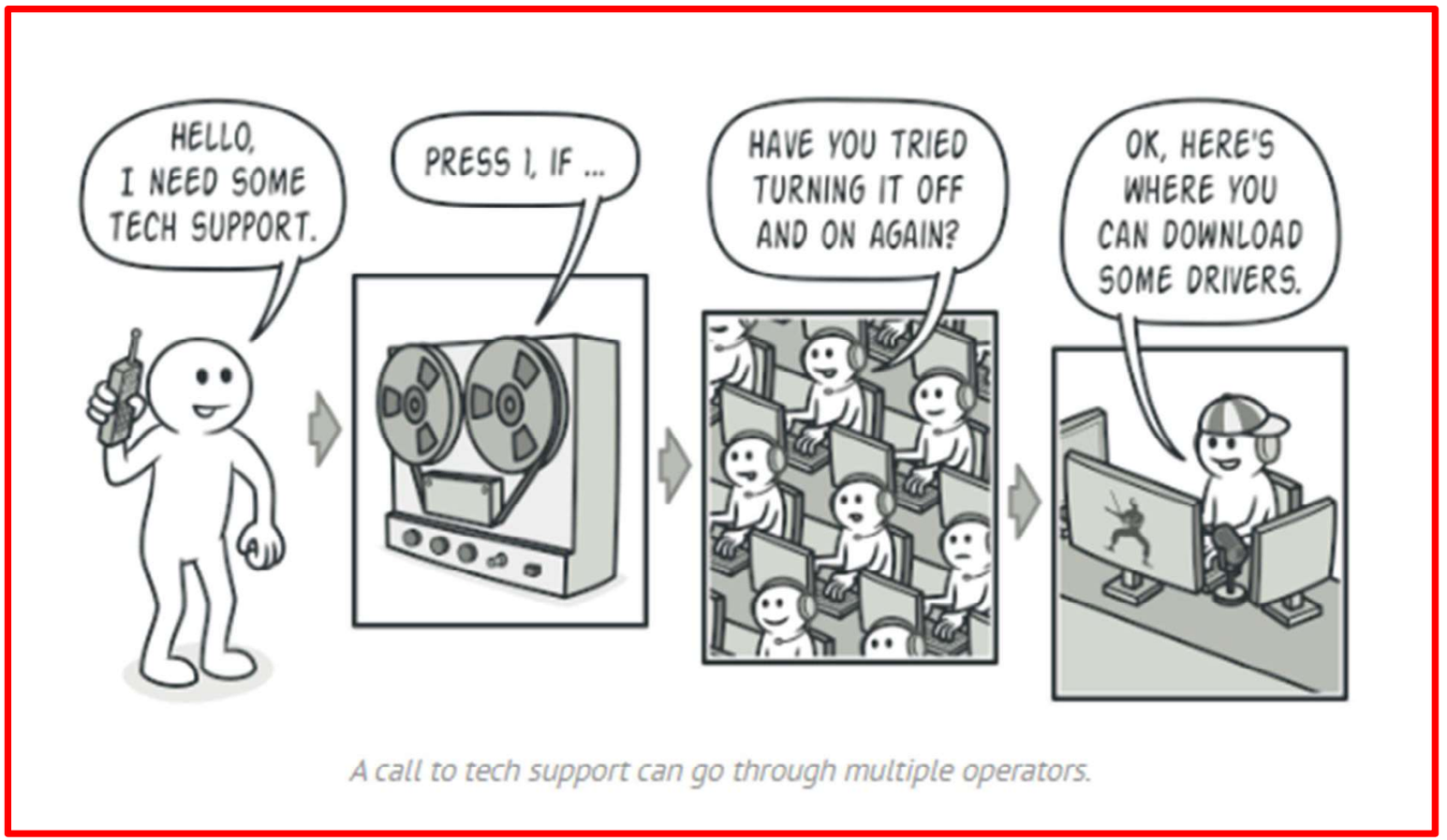
- Like many other behavioral design patterns, the **Chain of Responsibility** relies on transforming particular behaviors into stand-alone objects called *handlers*. In our case, each check should be extracted to its own class with a single method that performs the check. The request, along with its data, is passed to this method as an argument.

- The pattern suggests that you link these handlers into a chain. Each linked handler has a field for storing a reference to the next handler in the chain. In addition to processing a request, handlers pass the request further along the chain. The request travels along the chain until all handlers have had a chance to process it.

- Here's the best part: a handler can decide not to pass the request further down the chain and effectively stop any further processing.


Chain of Responsibility
Request
Handler
Handler
Handler
Ordering System
Handlers are lined up one by one, forming a chain.

Real-World Analogy



A call to tech support can go through multiple operators.

Complexity: ★★☆

Popularity: ★★☆

Usage examples: The Chain of Responsibility is pretty common in Java.

One of the most popular use cases for the pattern is bubbling events to the parent components in GUI classes. Another notable use case is sequential access filters.

Here are some examples of the pattern in core Java libraries:

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

Example

```
public class Currency {

    private int amount;

    public Currency(int amt) {
        this.amount = amt;
    }

    public int getAmount() {
        return amount;
    }

}
```

Example

```
package training.iqgateway.chainofresponsibility;

/**
 *
 * @author Sai Baba
 */
public interface DispenseChain {

    public void setNextChain(DispenseChain nextChain);

    public void dispense(Currency cur);

}
```

Example

```
public class Rupee500Dispenser implements DispenseChain {

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain = nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 500) {
            int num = cur.getAmount() / 500;
            int remainder = cur.getAmount() % 500;
            System.out.println("Dispensing " + num + " 500 Notes ");
            if(remainder != 0 ) {
                this.chain.dispense(new Currency(remainder));
            }
        }
```

```
else {
            this.chain.dispense(cur);
        }
    }
```

Example

```
public class Rupee200Dispenser implements DispenseChain {

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain = nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 200) {
            int num = cur.getAmount() / 200;
            int remainder = cur.getAmount() % 200;
            System.out.println("Dispensing " + num + " 200 Notes ");
            if(remainder != 0 ) {
                this.chain.dispense(new Currency(remainder));
            }
        }
```

```
else {
            this.chain.dispense(cur);
        }
    }
```

Example

```
public class Rupee100Dispenser implements DispenseChain {

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain = nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 100) {
            int num = cur.getAmount() / 100;
            int remainder = cur.getAmount() % 100;
            System.out.println("Dispensing " + num + " 100 Notes ");
            if(remainder != 0 ) {
                this.chain.dispense(new Currency(remainder));
            }
        }
```

```
else {
            this.chain.dispense(cur);
        }
    }
```

Example

```
public class CORTester {

    private DispenseChain c1;

    public CORTester() {
        // Initialize the Chain
        this.c1 = new Rupee500Dispenser();
        DispenseChain c2 = new Rupee200Dispenser();
        DispenseChain c3 = new Rupee100Dispenser();

        // Setting the Chain of Responsibility
        c1.setNextChain(c2);
        c2.setNextChain(c3);

    }
```

Example

```
public static void main(String[] args) {
        CORTester corTesterRef = new CORTester();
        while(true) {
            int amount = 0;
            System.out.println("Enter Withdrawl Amount : ");
            java.util.Scanner scanObj = new java.util.Scanner(System.in);
            amount = scanObj.nextInt();
            if(amount % 100 != 0) {
                System.out.println("Amount Should be in Multiples of 100's");
            }
            else {
                corTesterRef.c1.dispense(new Currency(amount));
            }
            return;
        }
    }
}
```

Applicability

1. Use the Chain of Responsibility pattern when your program is expected to process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand.

2. Use the pattern when it’s essential to execute several handlers in a particular order.

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.

Command Pattern

Introduction

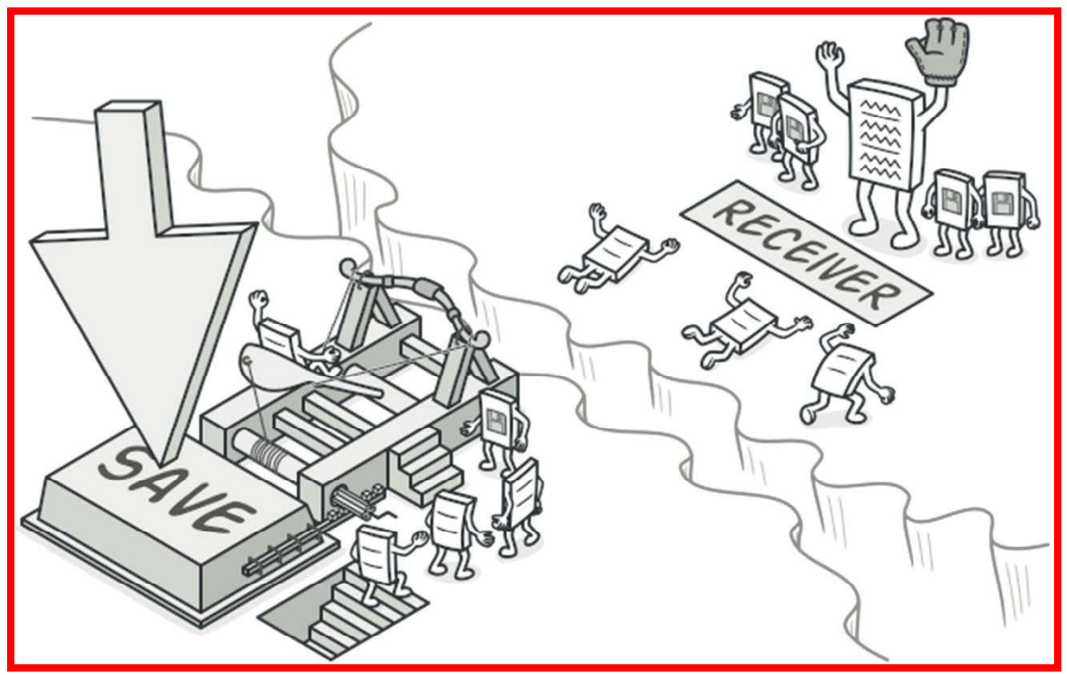
- ***Command*** *is behavioral design pattern that converts requests or simple operations into objects.*

- *The conversion allows deferred or remote execution of commands, storing command history, etc.*

- *The command pattern is used to implement loose-coupling in a request-response model. In this pattern, the request is sent to the invoker and the invoker passes it to the encapsulated command object. The command object passes the request to the appropriate method of receiver to perform the specific action.*

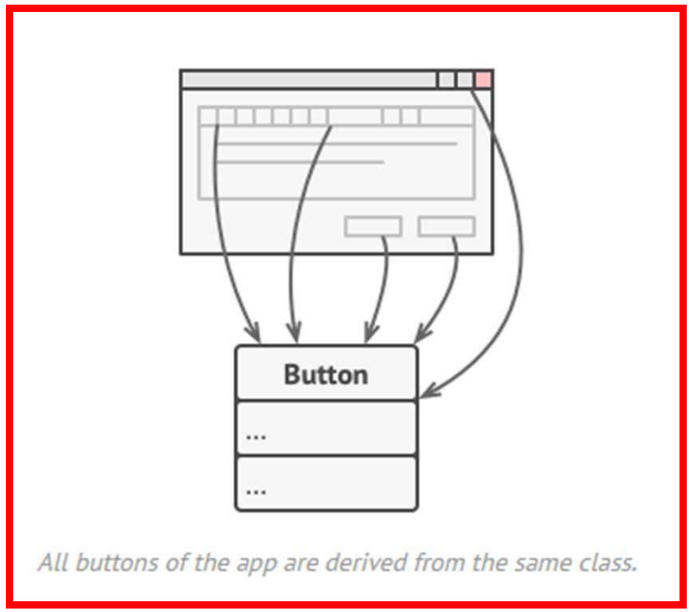
Intent

- **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 pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.



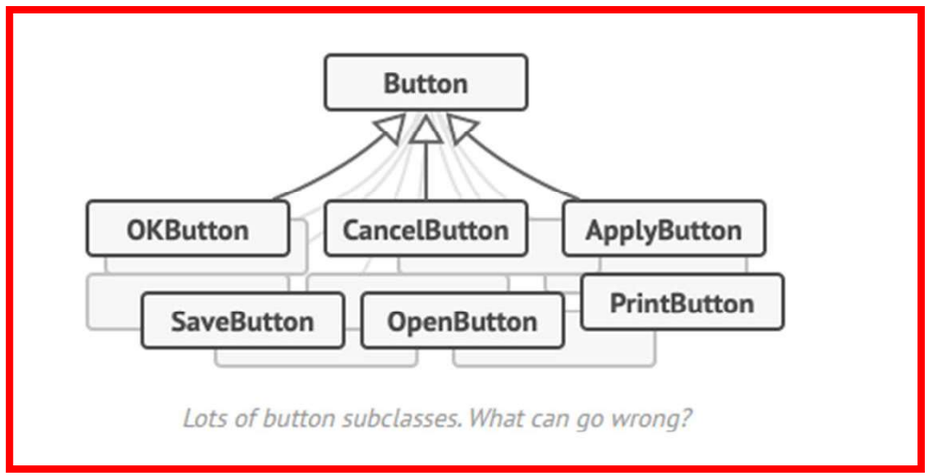
Problem

- Imagine that you're working on a new text-editor app. Your current task is to create a toolbar with a bunch of buttons for various operations of the editor. You created a very neat Button class that can be used for buttons on the toolbar, as well as for generic buttons in various dialogs.



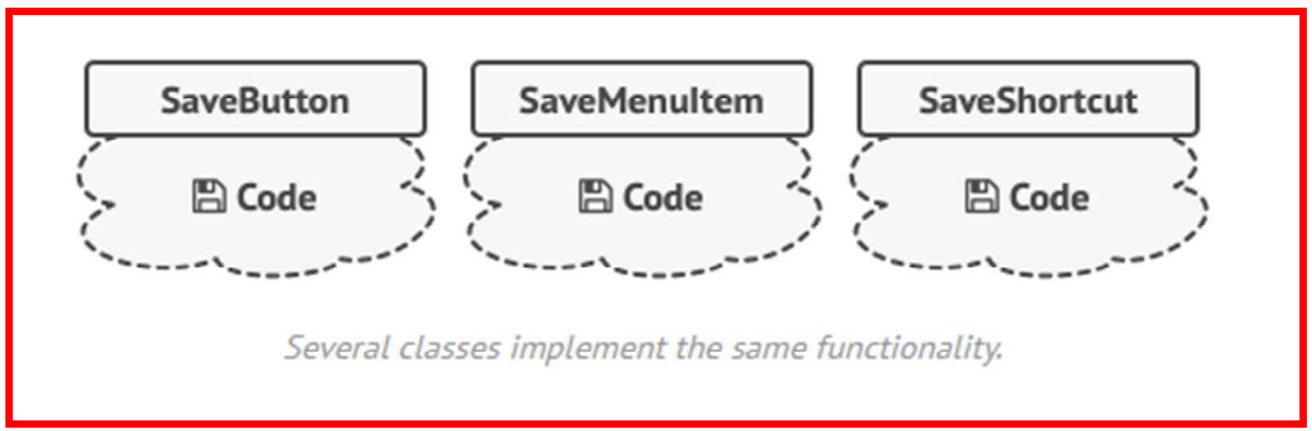
All buttons of the app are derived from the same class.

- While all of these buttons look similar, they're all supposed to do different things. Where would you put the code for the various click handlers of these buttons? The simplest solution is to create tons of subclasses for each place where the button is used. These subclasses would contain the code that would have to be executed on a button click.



Lots of button subclasses. What can go wrong?

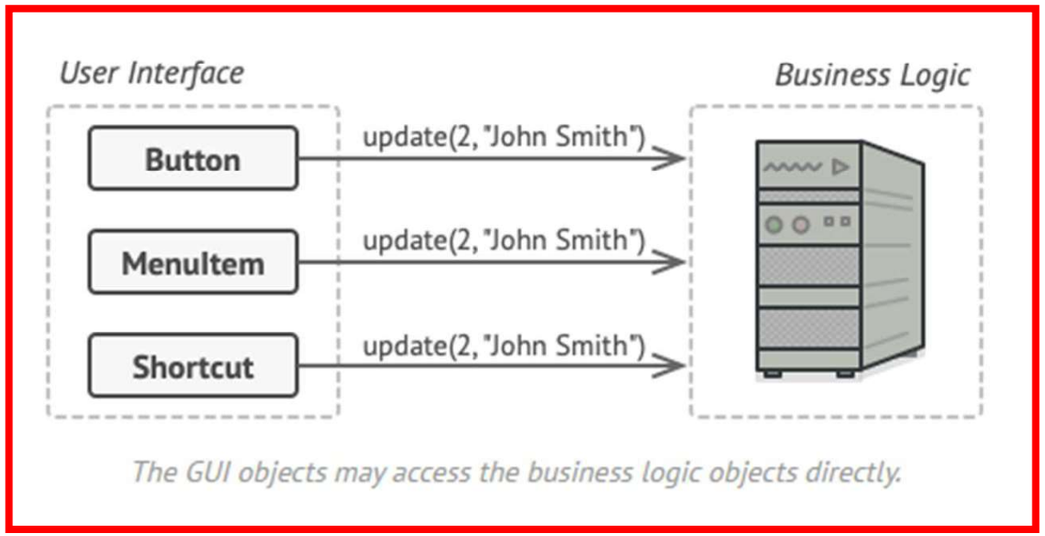
- Before long, you realize that this approach is deeply flawed. First, you have an enormous number of subclasses, and that would be okay if you weren't risking breaking the code in these subclasses each time you modify the base Button class. Put simply, your GUI code has become awkwardly dependent on the volatile code of the business logic.



Several classes implement the same functionality.

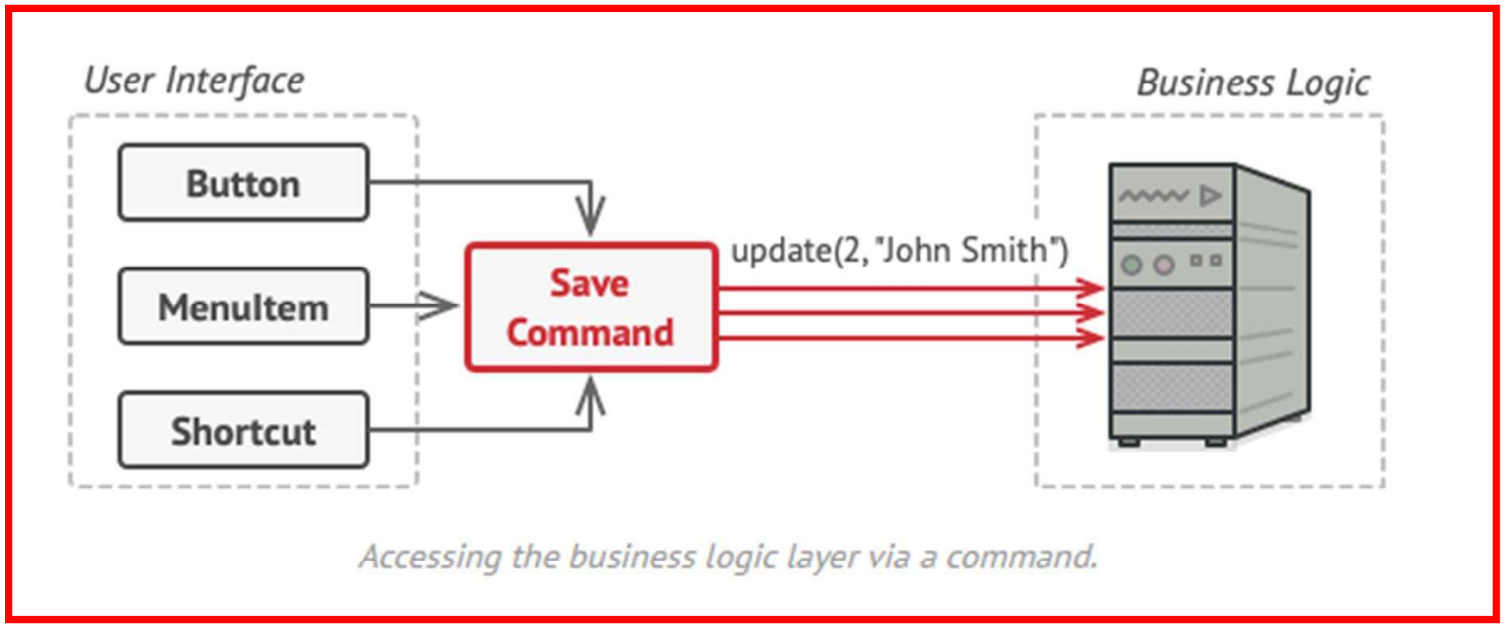
- Good software design is often based on the ***principle of separation of concerns***, which usually results in breaking an app into layers.

- A GUI object calls a method of a business logic object, passing it some arguments. This process is usually described as one object sending another a *request*.

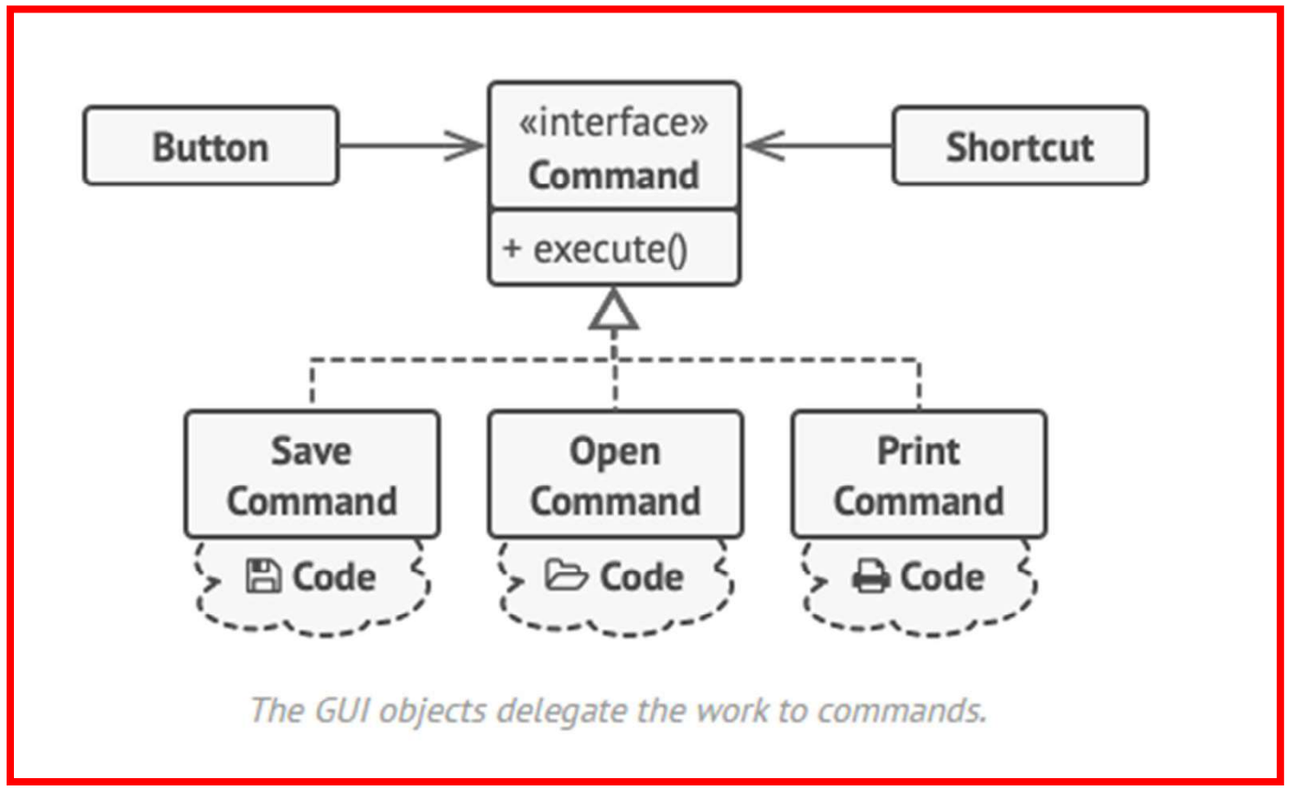


- The Command pattern suggests that GUI objects shouldn't send these requests directly. Instead, you should extract all of the request details, such as the object being called, the name of the method and the list of arguments into a separate *command* class with a single method that triggers this request.

- Command objects serve as links between various GUI and business logic objects. From now on, the GUI object doesn't need to know what business logic object will receive the request and how it'll be processed. The GUI object just triggers the command, which handles all the details.

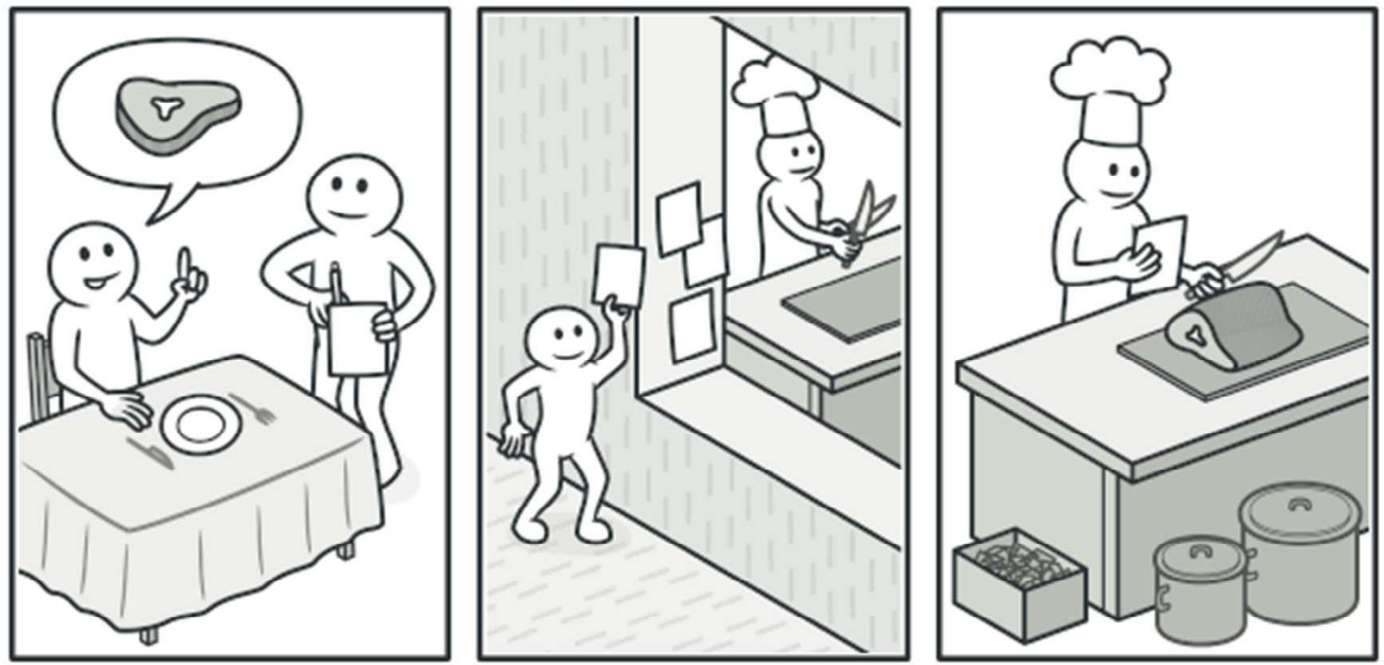


Accessing the business logic layer via a command.



The GUI objects delegate the work to commands.

Real-World Analogy



Making an order in a restaurant.

Complexity: ★☆☆

Popularity: ★★★

Usage examples: The Command pattern is pretty common in Java code. Most often it's used as an alternative for callbacks to parameterizing UI elements with actions. It's also used for queueing tasks, tracking operations history, etc.

Here are some examples of Commands in core Java libraries:

- All implementations of `java.lang.Runnable`
- All implementations of `javax.swing.Action`

Example

```
// Driver code
    public static void main(String args[])
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;

        // Function call
        int result = search(arr, x);
        if (result == -1)
            System.out.print("Element is not present in array");
        else
            System.out.print("Element is present at index “ + result);
    }
}
```

Applicability

1. Use the Command pattern when you want to parametrize objects with operations.

2. Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely.

3. Use the Command pattern when you want to implement reversible operations. [Undo/Redo]

Pros and Cons

- ✓ *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.*

- ✗ The code may become more complicated since you're introducing a whole new layer between senders and receivers.

Iterator Pattern

Introduction

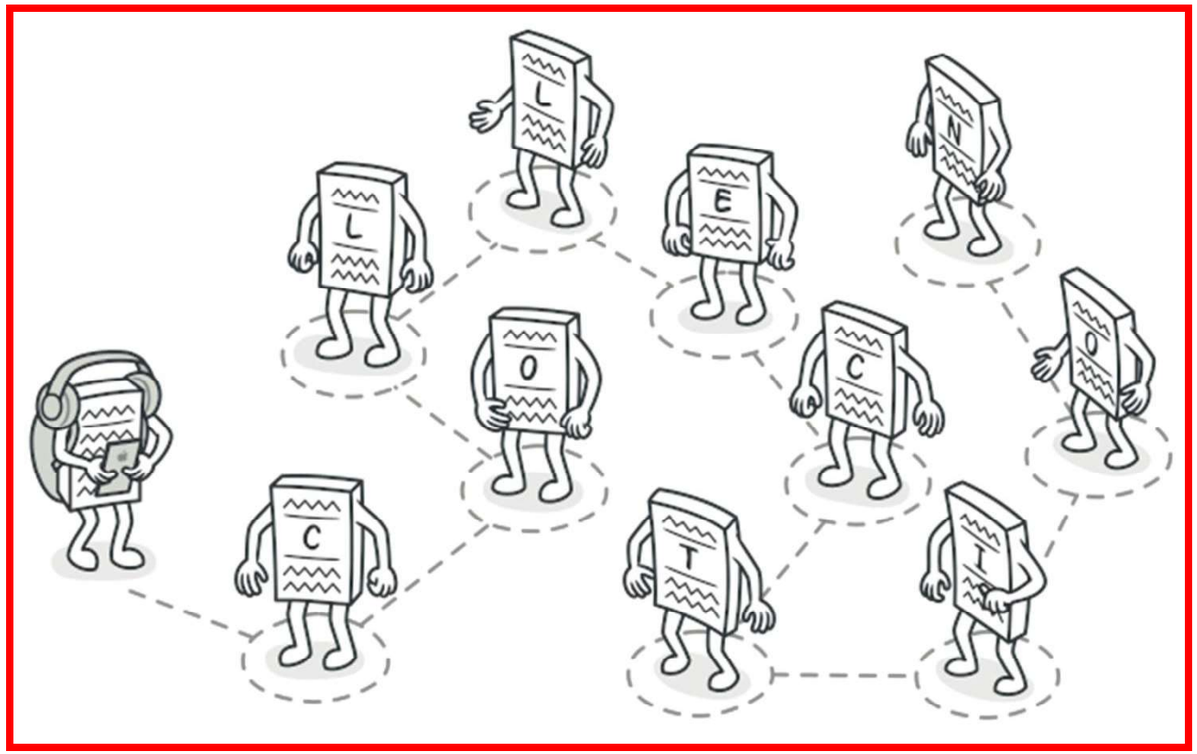
- ***Iterator*** *is a behavioral design pattern that allows sequential traversal through a complex data structure without exposing its internal details.*

- Thanks to the Iterator, clients can go over elements of different collections in a similar fashion using a single iterator interface.

- *The iterator pattern is one of the behavioral patterns and is used to provide a standard way to traverse through a group of objects. The iterator pattern is widely used in* ***Java Collection Framework*** *where the iterator interface provides methods for traversing through a Collection. This pattern is also used to provide different kinds of iterators based on our requirements. The iterator pattern hides the actual implementation of traversal through the Collection and client programs use iterator methods.*

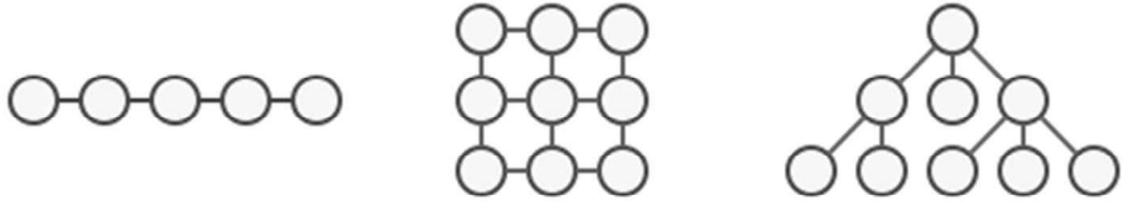
Intent

- **Iterator** is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



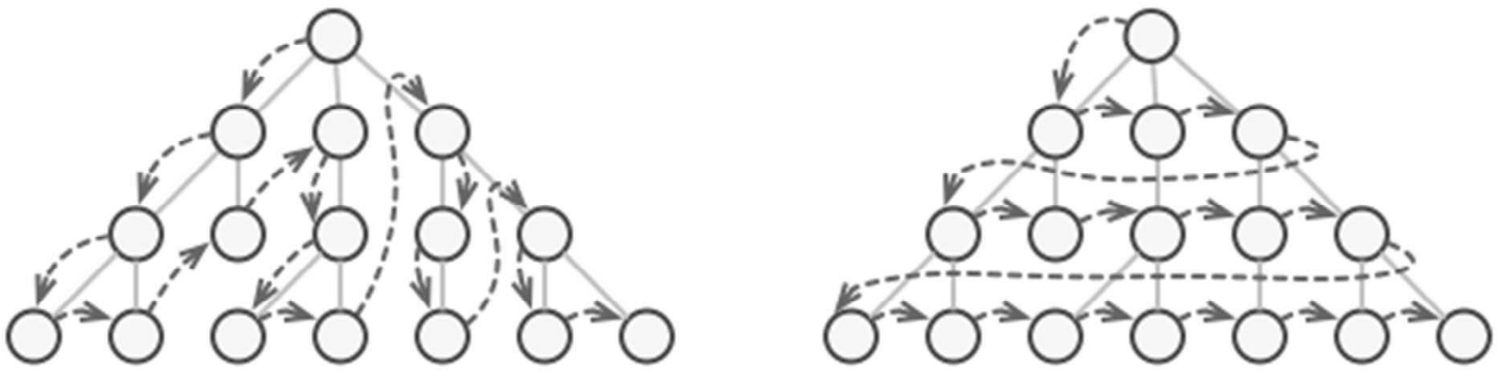
Problem

- Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.



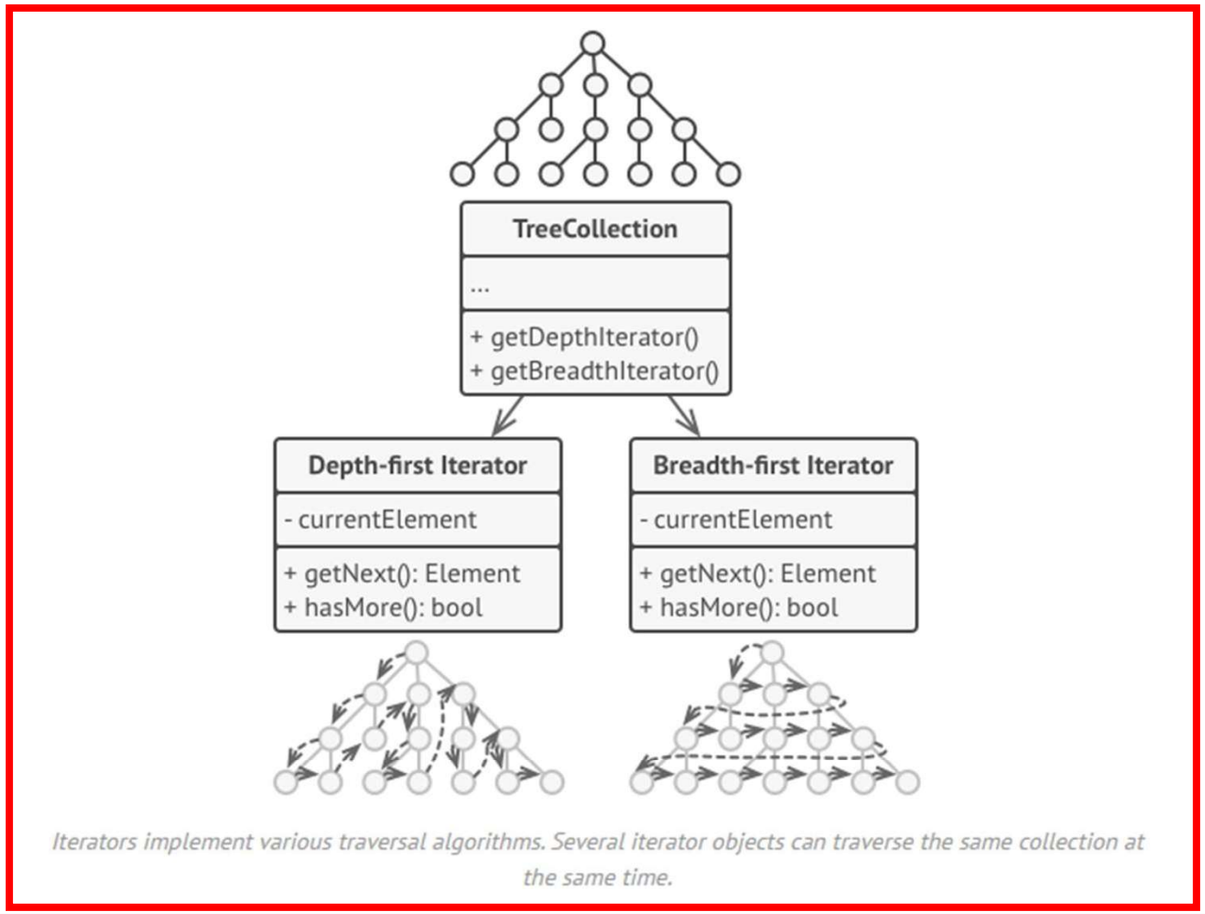
Various types of collections.

- Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.
- But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements.



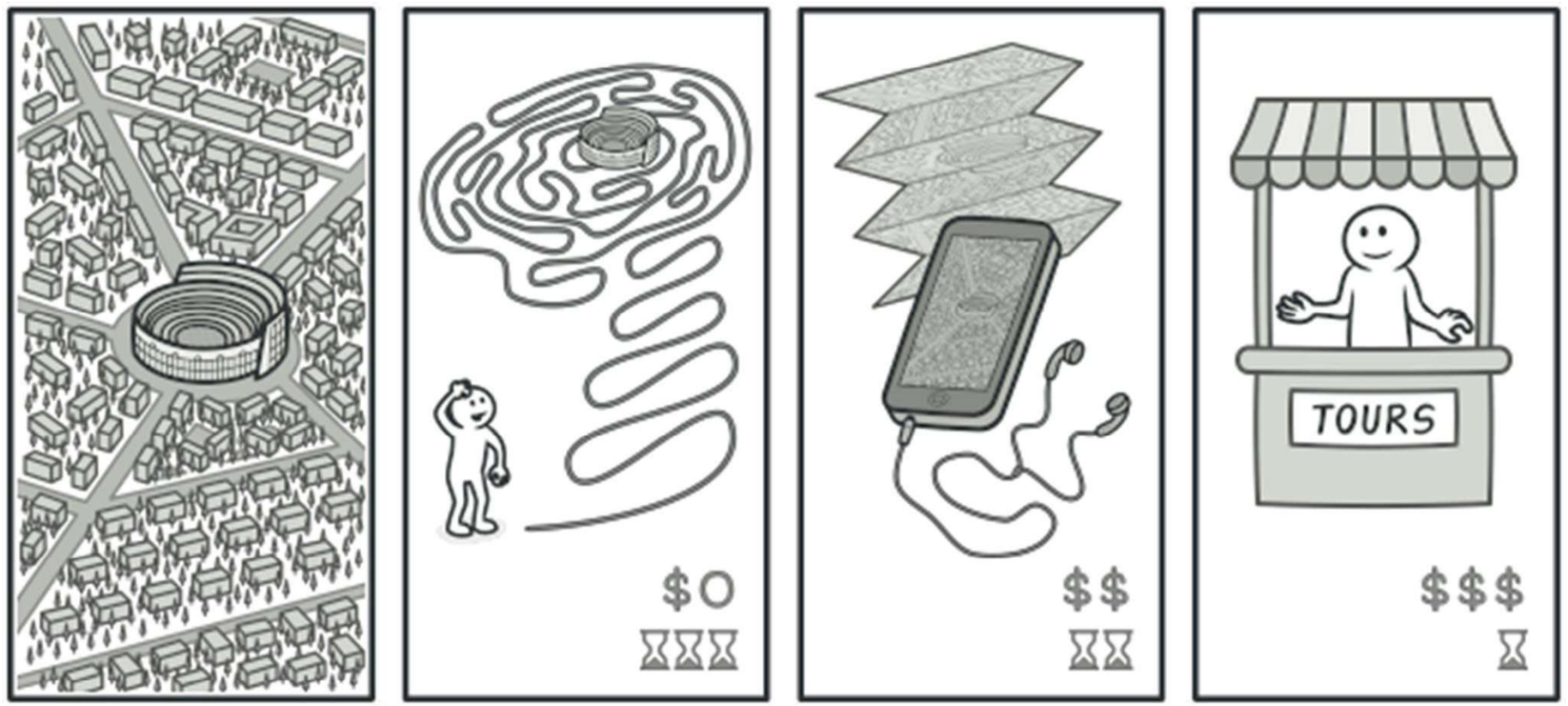
The same collection can be traversed in several different ways.

- The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an **Iterator**.



Iterators implement various traversal algorithms. Several iterator objects can traverse the same collection at the same time.

Real-World Analogy



Various ways to walk around Rome.

Complexity: ★★☆

Popularity: ★★★

Usage examples: The pattern is very common in Java code. Many frameworks and libraries use it to provide a standard way for traversing their collections.

Here are some examples from core Java libraries:

- All implementations of `java.util.Iterator` (also `java.util.Scanner`).
- All implementations of `java.util.Enumeration`.

Example

```
// Driver code
    public static void main(String args[])
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;

        // Function call
        int result = search(arr, x);
        if (result == -1)
            System.out.print("Element is not present in array");
        else
            System.out.print("Element is present at index “ + result);
    }
}
```

Applicability

1. Use the Iterator pattern when your collection has a complex data structure under the hood, but you want to hide its complexity from clients (either for convenience or security reasons).

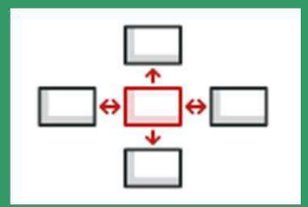
2. Use the pattern to reduce duplication of the traversal code across your app.

3. Use the Iterator when you want your code to be able to traverse different data structures or when types of these structures are unknown beforehand.

Pros and Cons

- ✓ *Single Responsibility Principle. You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes.*
- ✓ *Open/Closed Principle. You can implement new types of collections and iterators and pass them to existing code without breaking anything.*
- ✓ *You can iterate over the same collection in parallel because each iterator object contains its own iteration state.*
- ✓ *For the same reason, you can delay an iteration and continue it when needed.*

- ✘ Applying the pattern can be an overkill if your app only works with simple collections.
- ✘ Using an iterator may be less efficient than going through elements of some specialized collections directly.



Mediator Pattern