4

Behavioral Design Patterns

Objectives

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

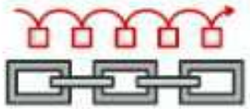
- Analysis of Behavioral DP
- Exploration of Chain of Responsibility Pattern, Command Pattern
- Iterator Pattern, Mediator Pattern
- State Pattern, Strategy Pattern, Template Pattern
- Memento Pattern, Visitor Pattern, Observer Pattern

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

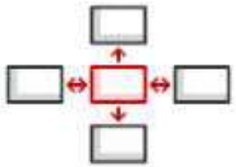
Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

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.

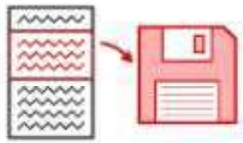
Iterator

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.

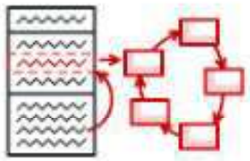


Memento

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

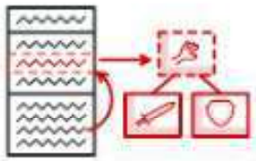
Observer

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



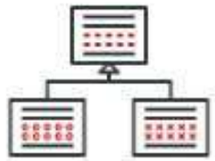
State

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



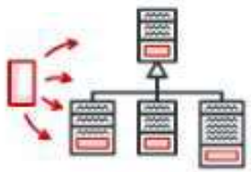
Strategy

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.



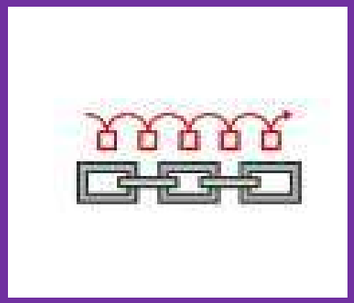
Visitor

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.*

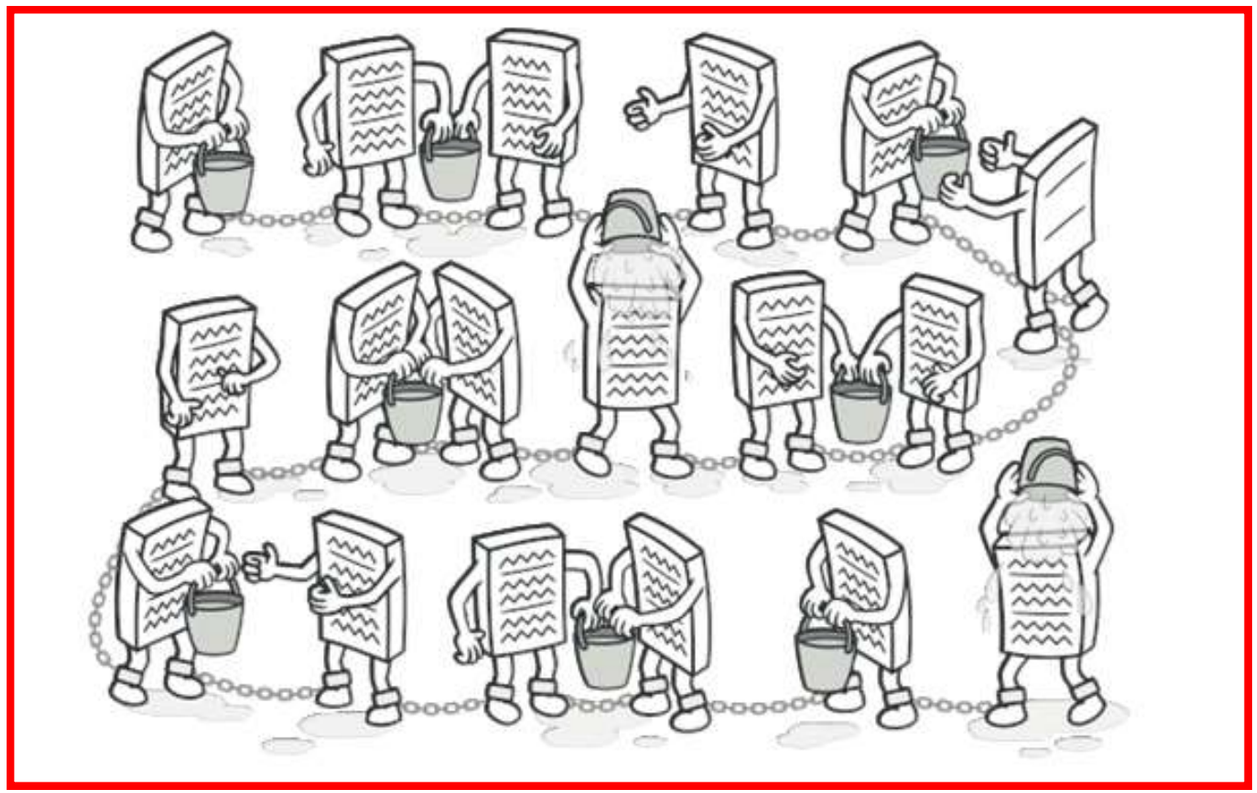
- *The chain of responsibility pattern is used to achieve loose-coupling in software design where a request from the client is passed to a chain of objects to process them. Then the object in the chain will decide who will be processing the request and whether the request is required to be sent to the 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.

Chain of Resposibility

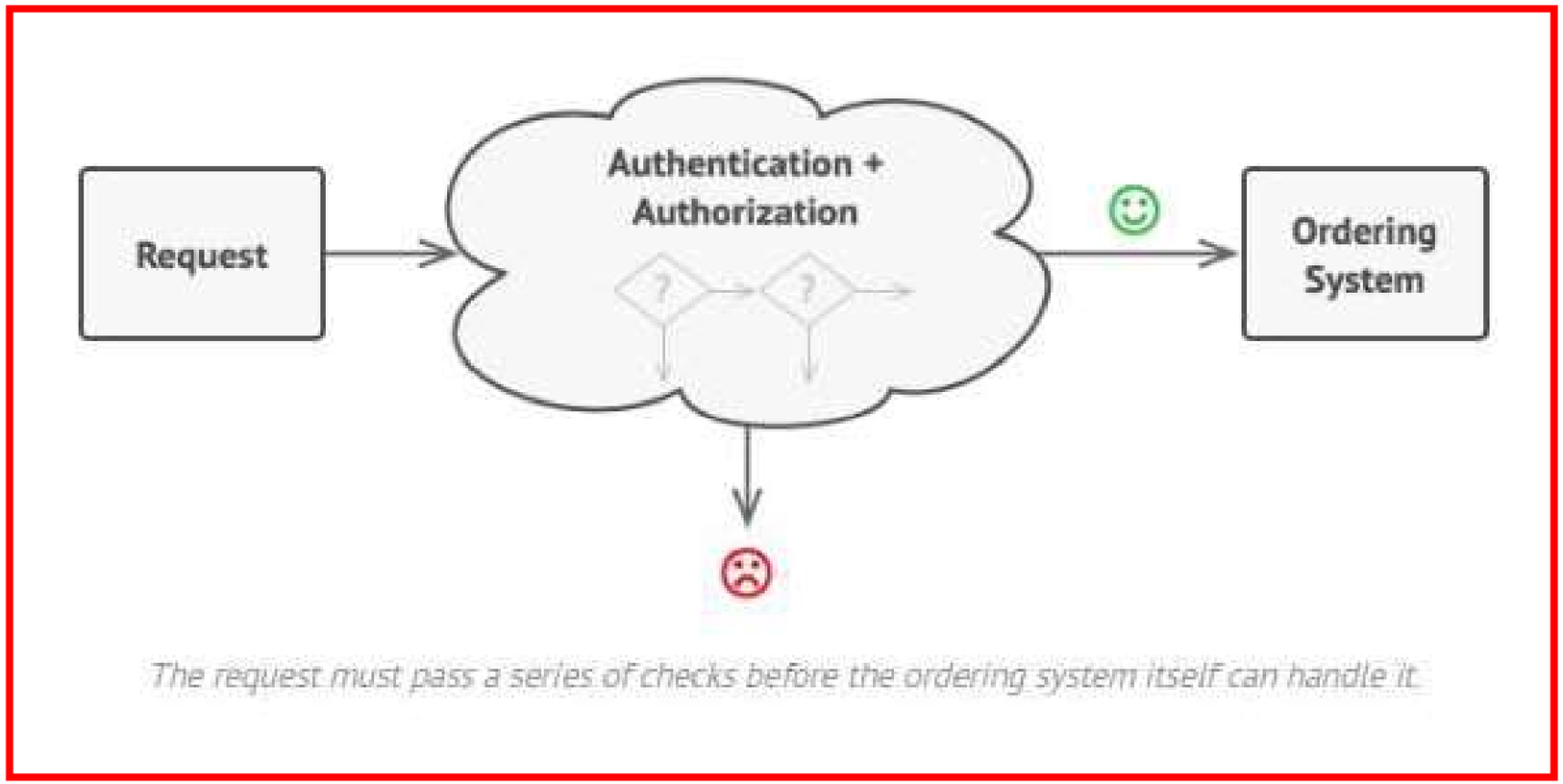
- **Intent**
  - **Chain of Responsibility** is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.



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.

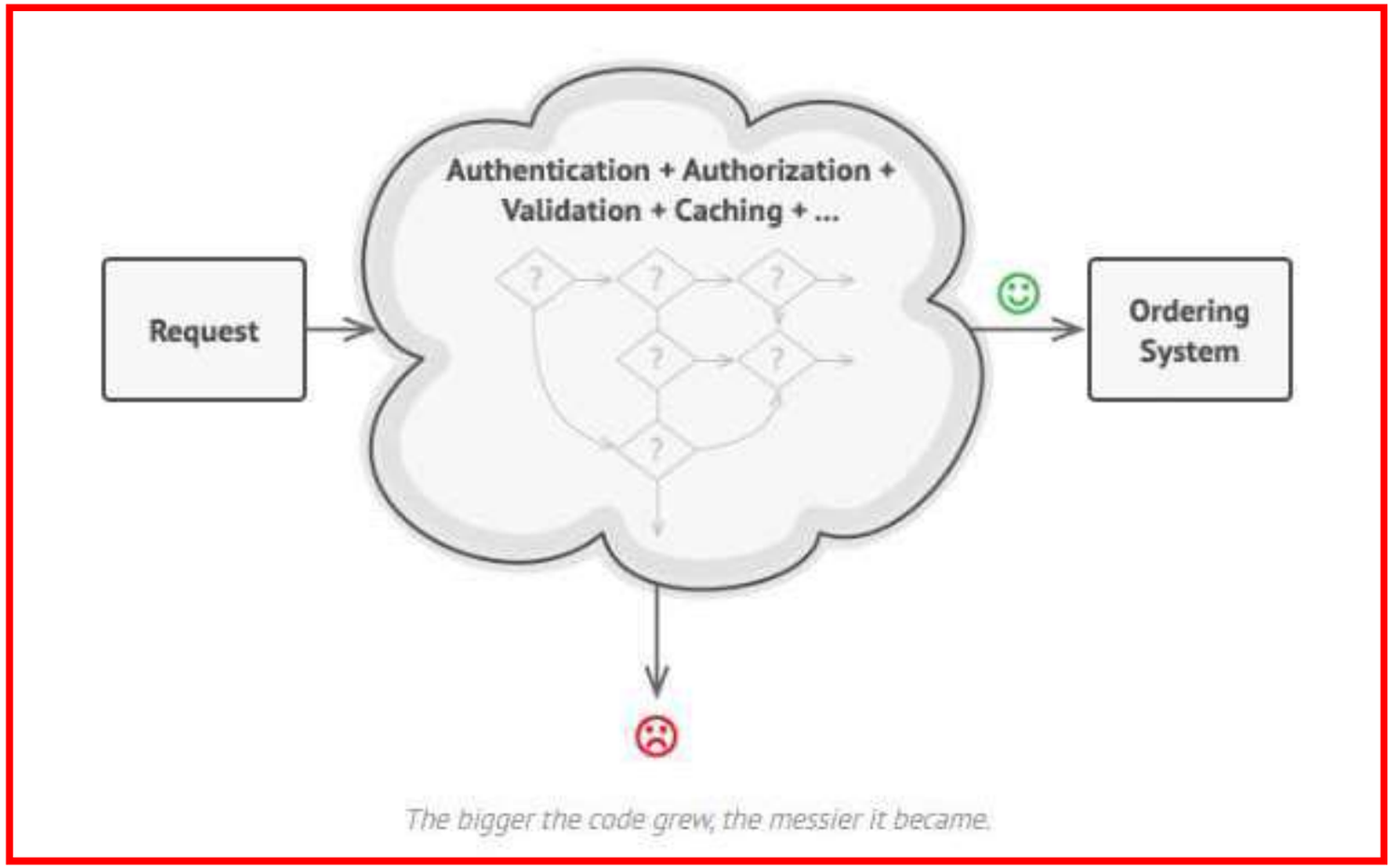
- After a bit of planning, you realized that these checks must be performed sequentially. The application can attempt to authenticate a user to the system whenever it receives a request that contains the user's credentials. However, if those credentials aren't correct and authentication fails, there's 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.

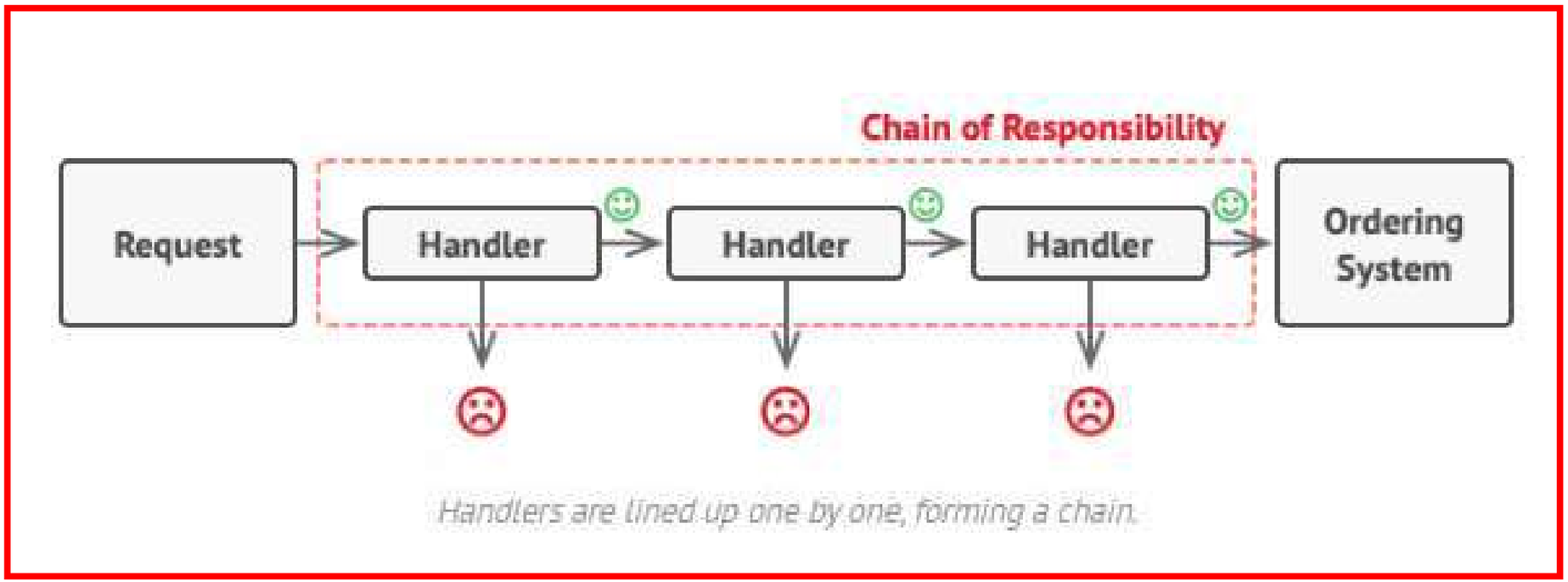
- One of your colleagues suggested that it's unsafe to pass raw data straight to the ordering system. So you added an extra validation step to sanitize the data in a request.
- Later, somebody noticed that the system is vulnerable to brute force password cracking. To negate this, you promptly added a 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.

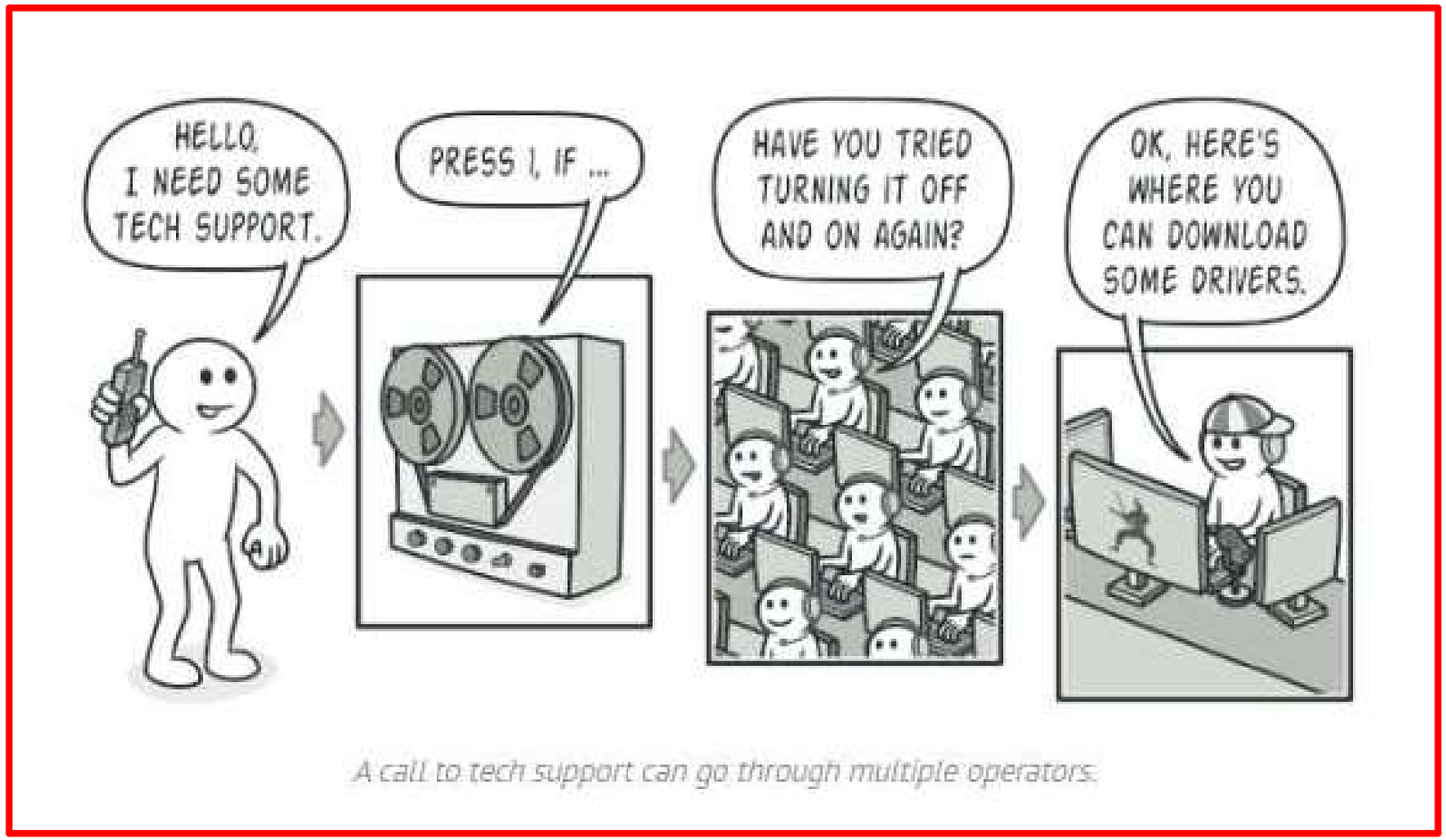
- Like many other behavioral design patterns, the **Chain of Responsibility** relies on transforming particular behaviors into stand-alone objects called *handlers*. In our case, each check should be extracted to its own class with a single method that performs the check. The request, along with its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

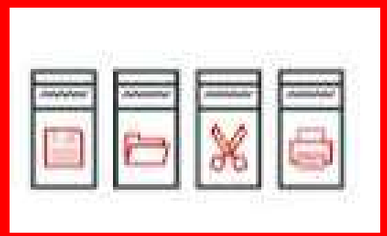
1. Use the Chain of Responsibility pattern when your program is expected to process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand.

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

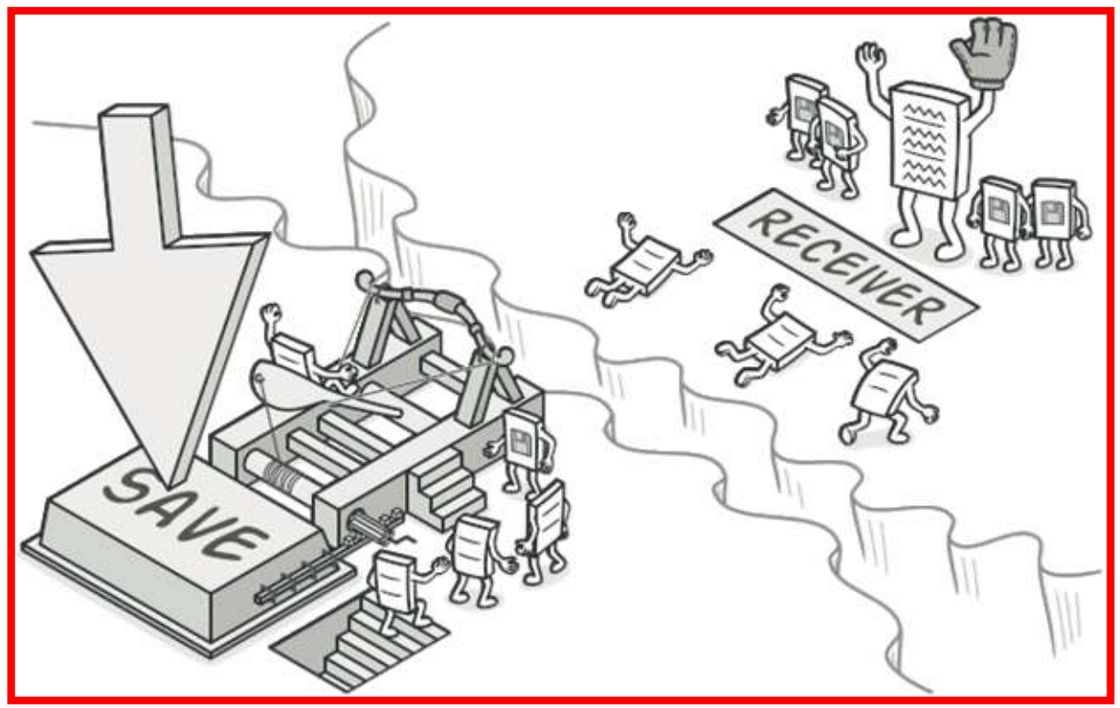
- ***Command*** *is behavioral design pattern that converts requests or simple operations into objects.*

- *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.*

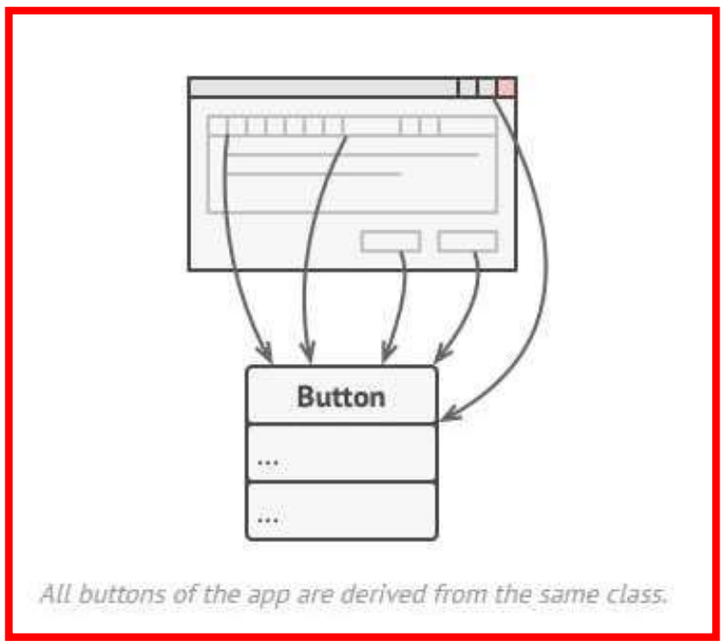
Intent

- **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.



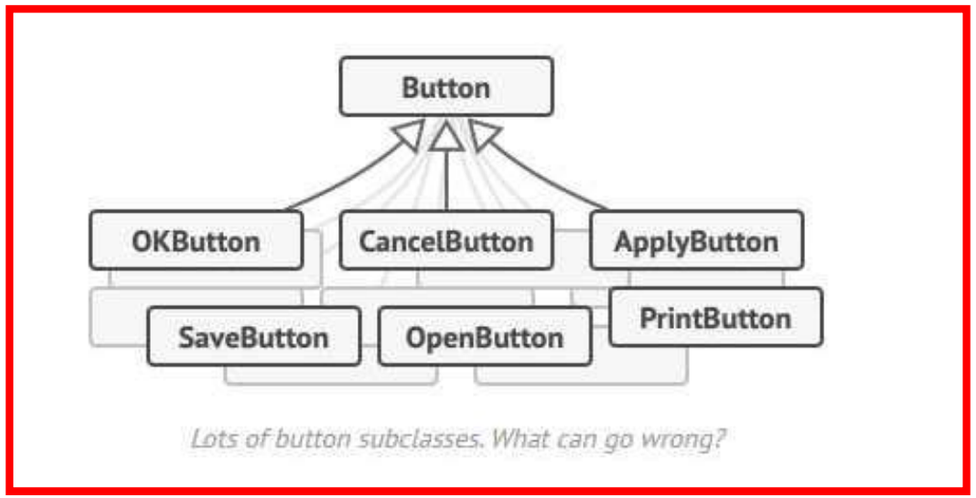
Problem

- 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.



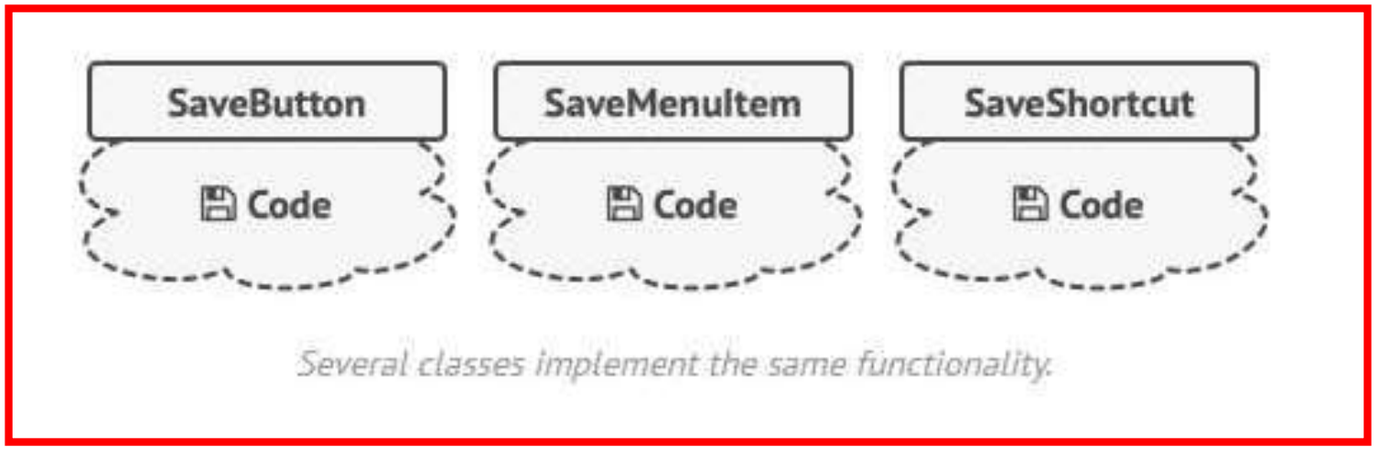
All buttons of the app are derived from the same class.

- 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?

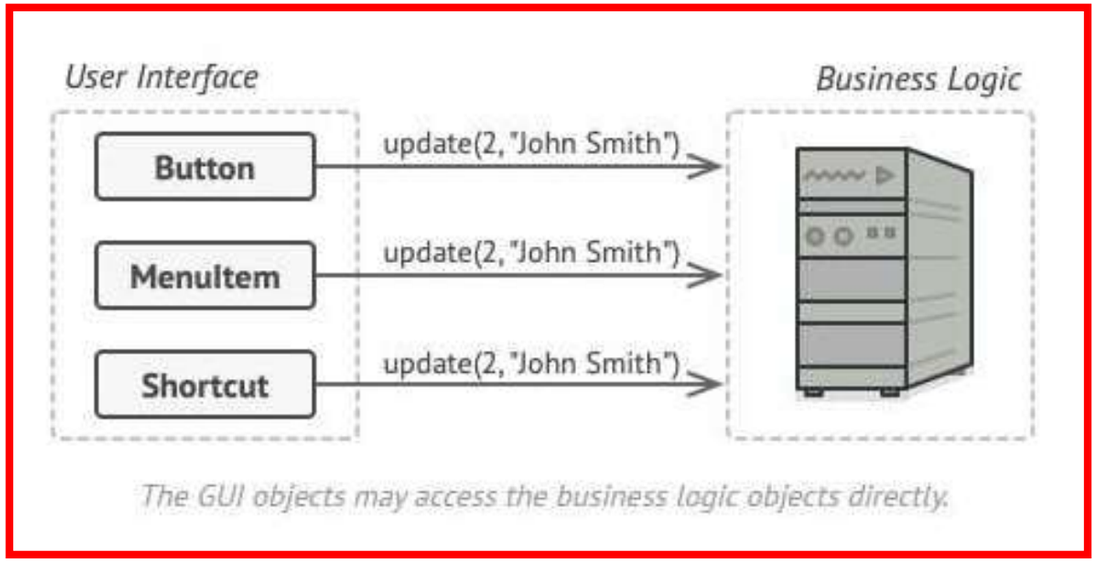
- Before long, you realize that this approach is deeply flawed. First, you have an enormous number of subclasses, and that would be okay if you weren't risking breaking the code in these subclasses each time you modify the base Button class. Put simply, your GUI code has become awkwardly dependent on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



Several classes implement the same functionality.

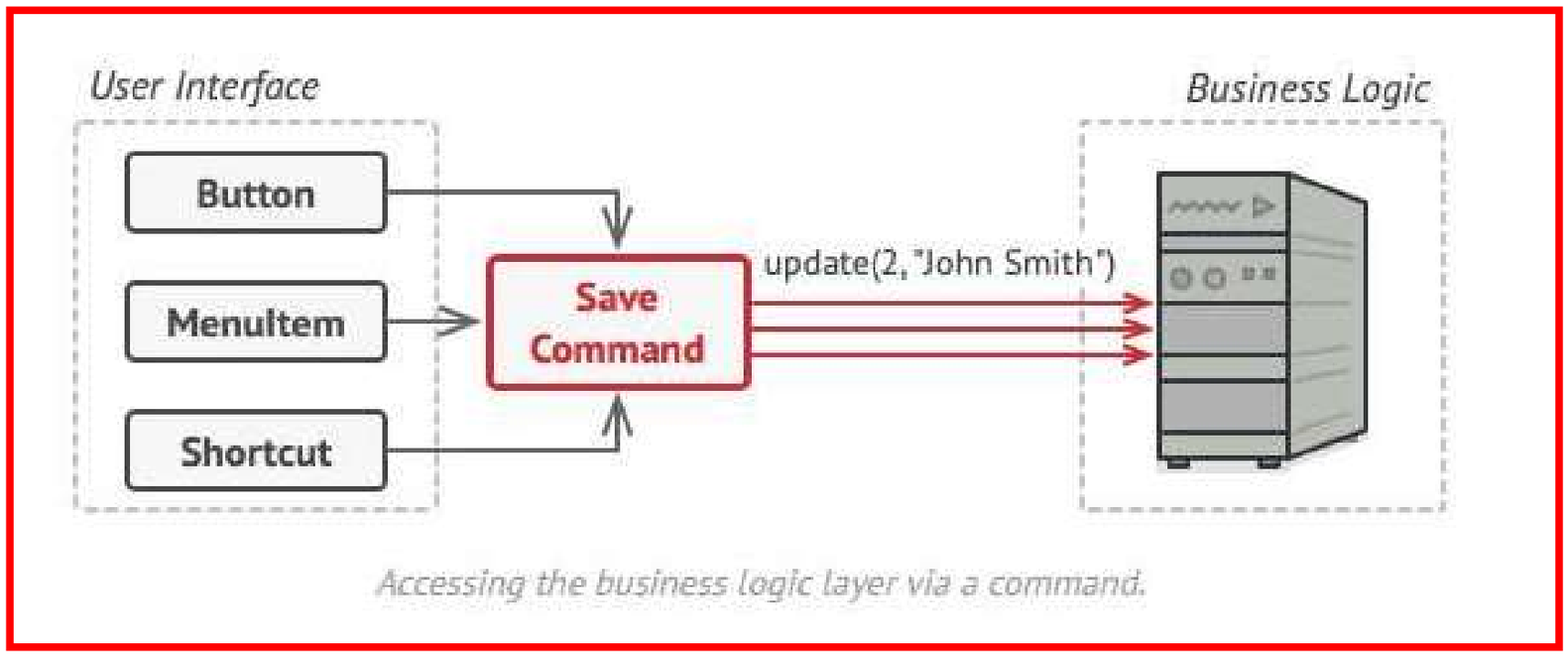
- Good software design is often based on the ***principle of separation of concerns***, which usually results in breaking an app into layers.

- 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*.

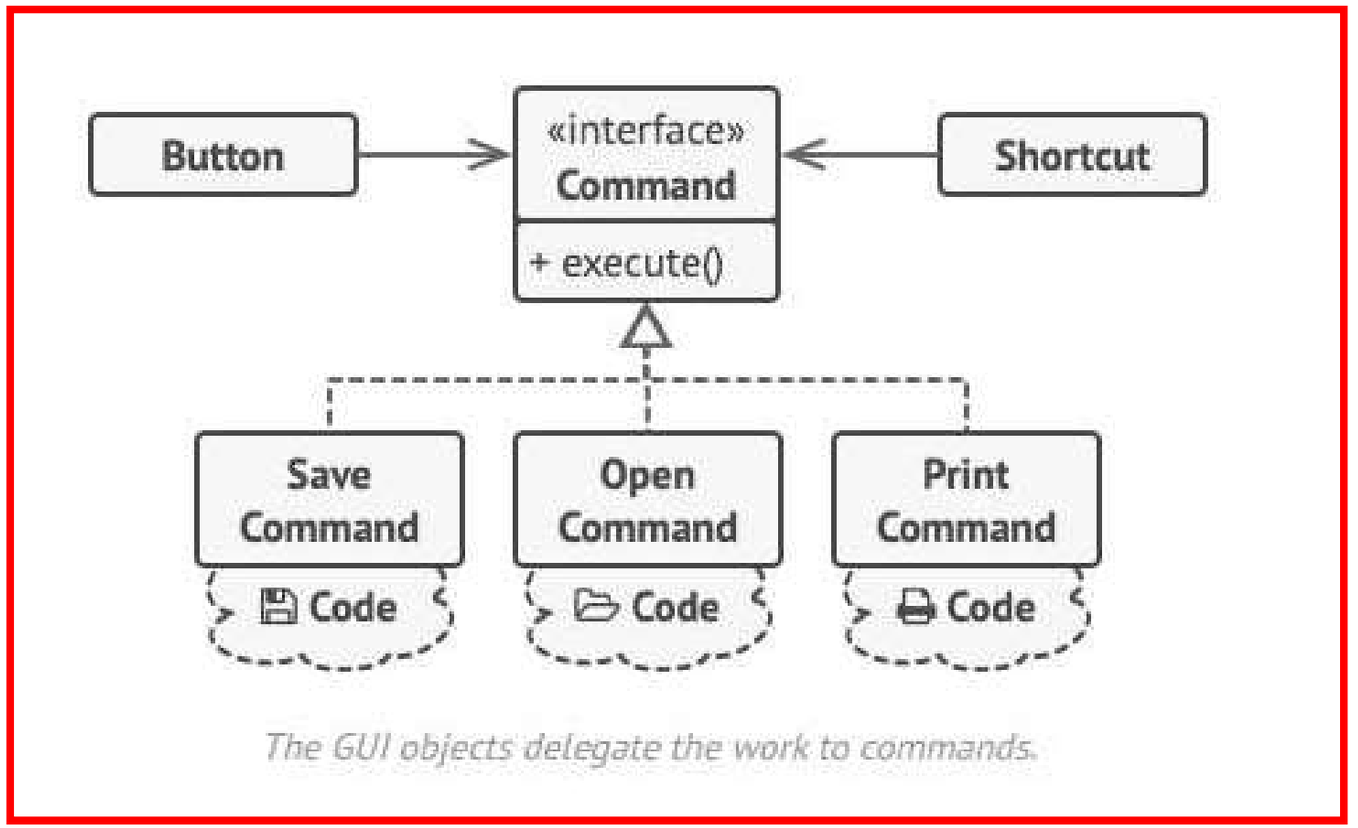


- The Command pattern suggests that GUI objects shouldn't send these requests directly. Instead, you should extract all of the request details, such as the object being called, the name of the method and the list of arguments into a separate *command* class with a single method that triggers this request.

- 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.

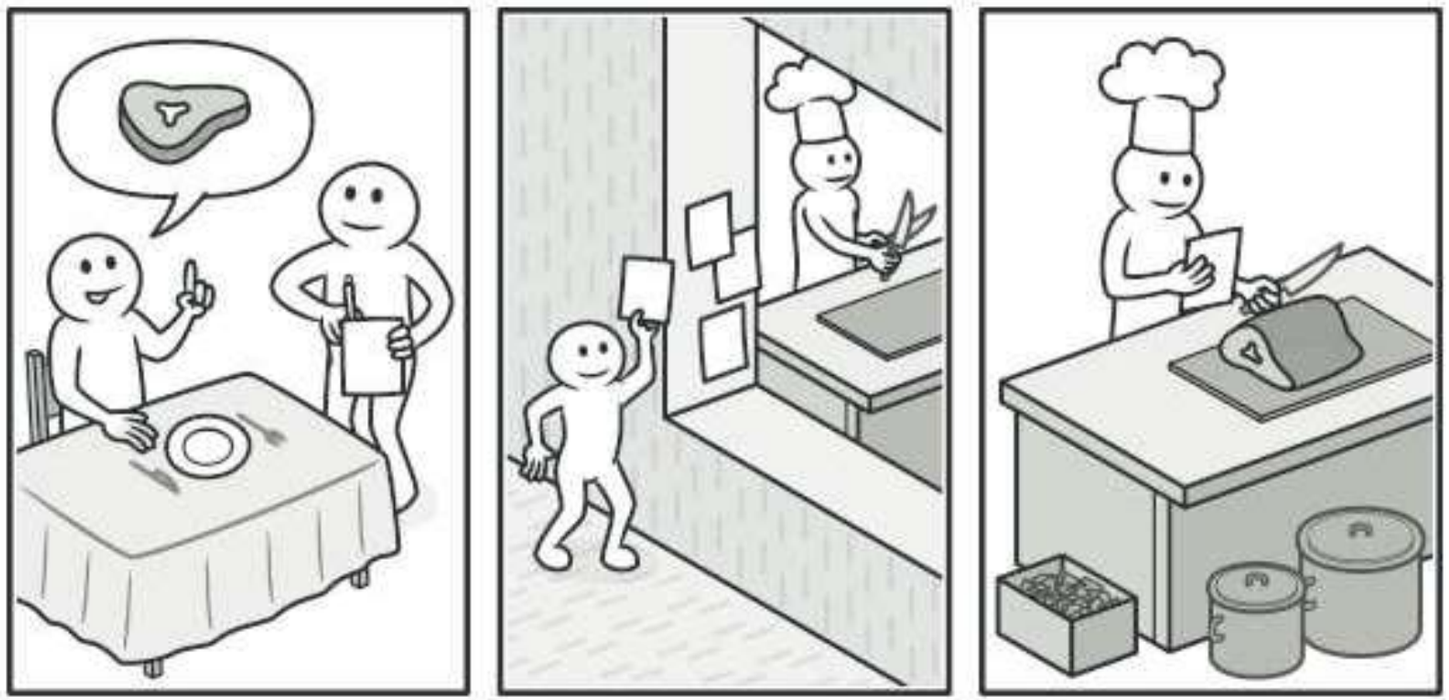


Accessing the business logic layer via a command.



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

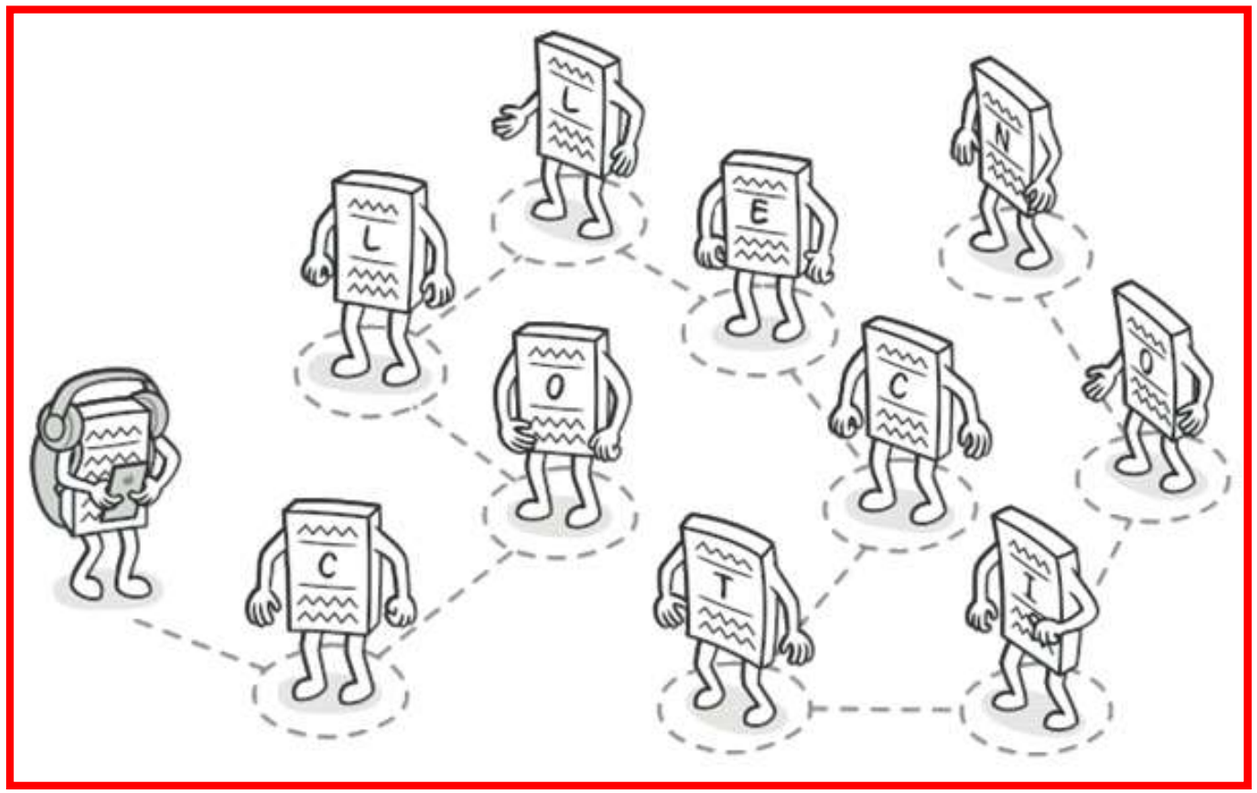
- ***Iterator*** *is a behavioral design pattern that allows sequential traversal through a complex data structure without exposing its internal details.*

- 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.*

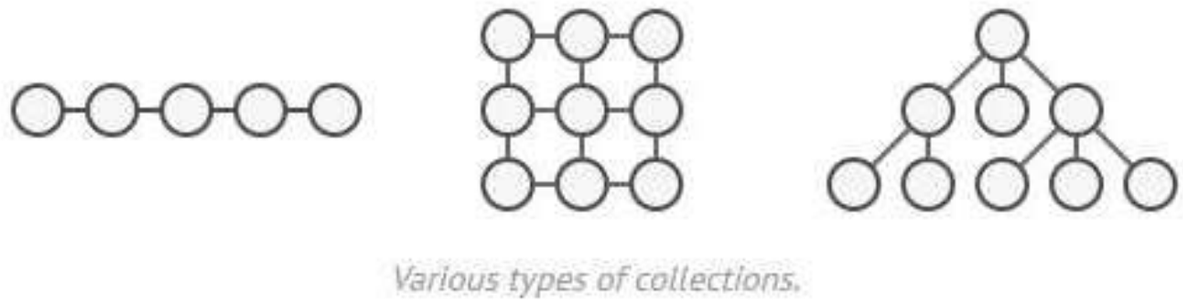
Intent

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



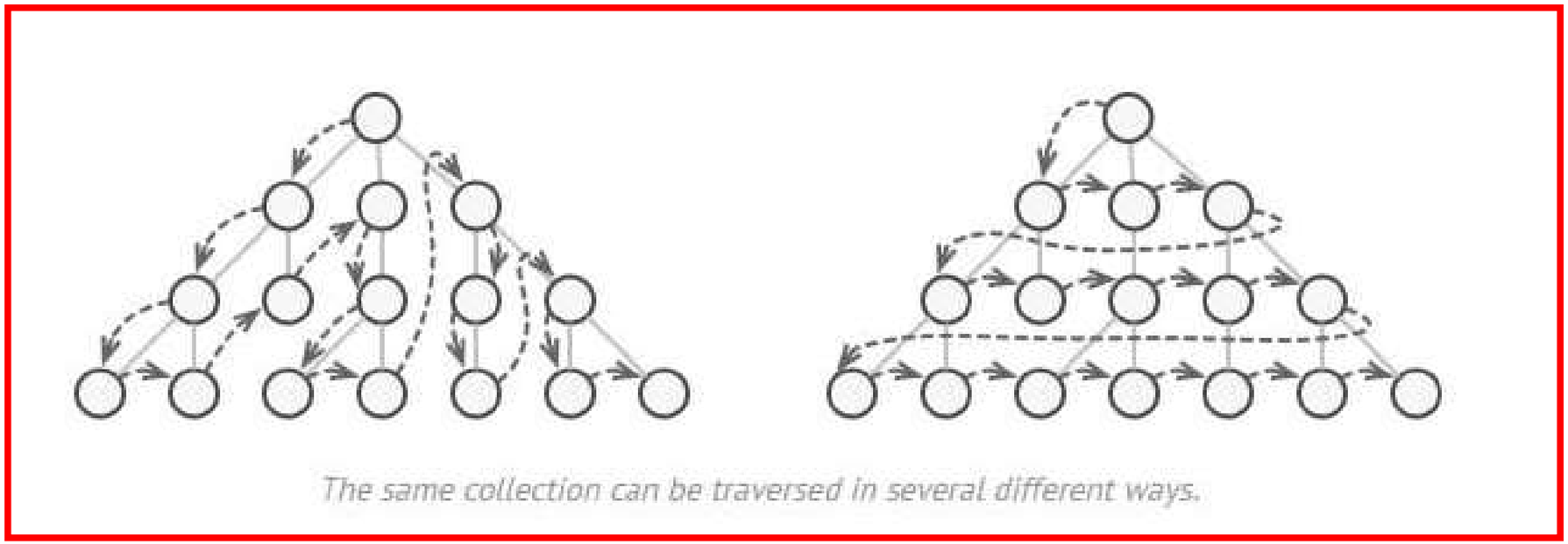
Problem

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



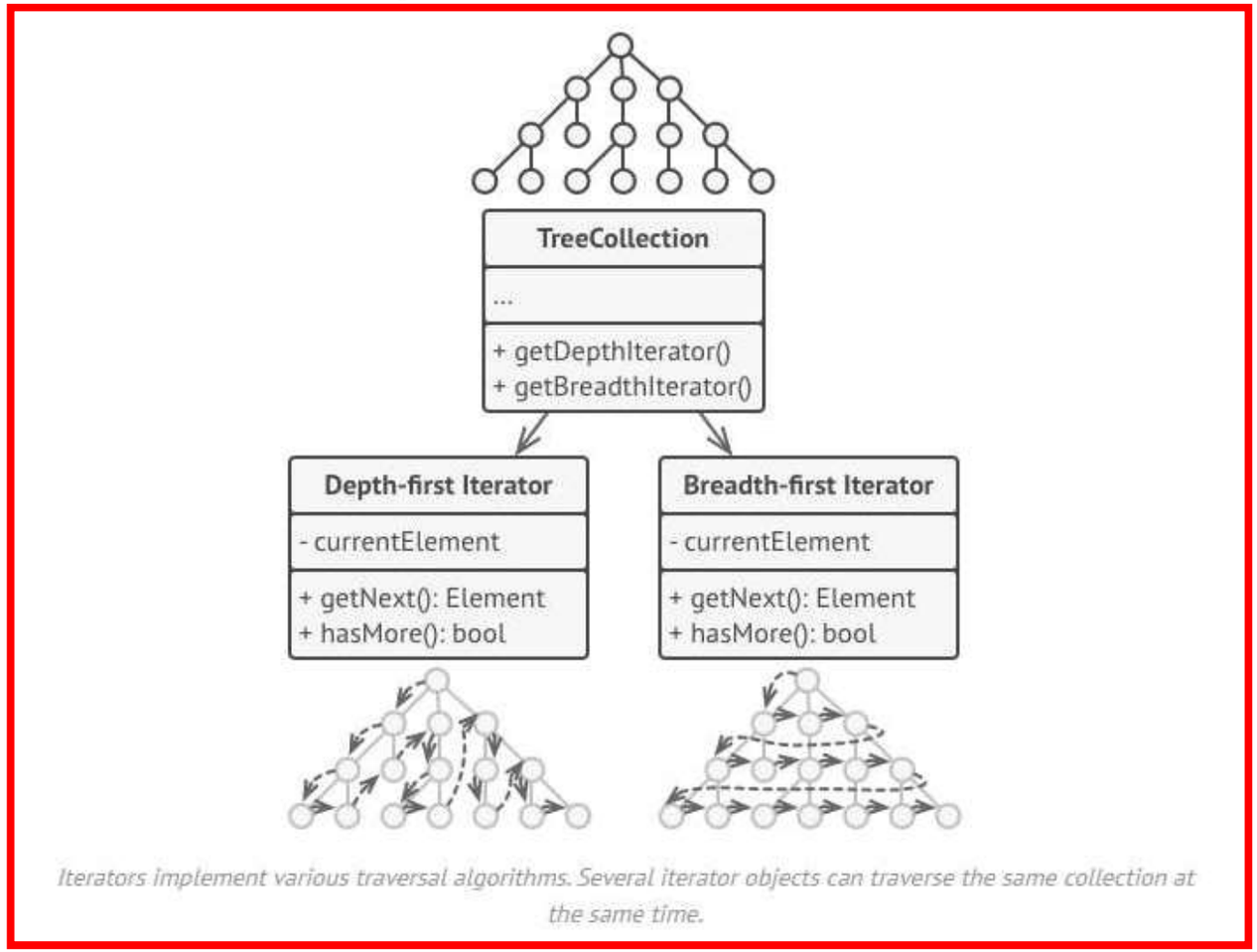
Various types of collections.

- 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.



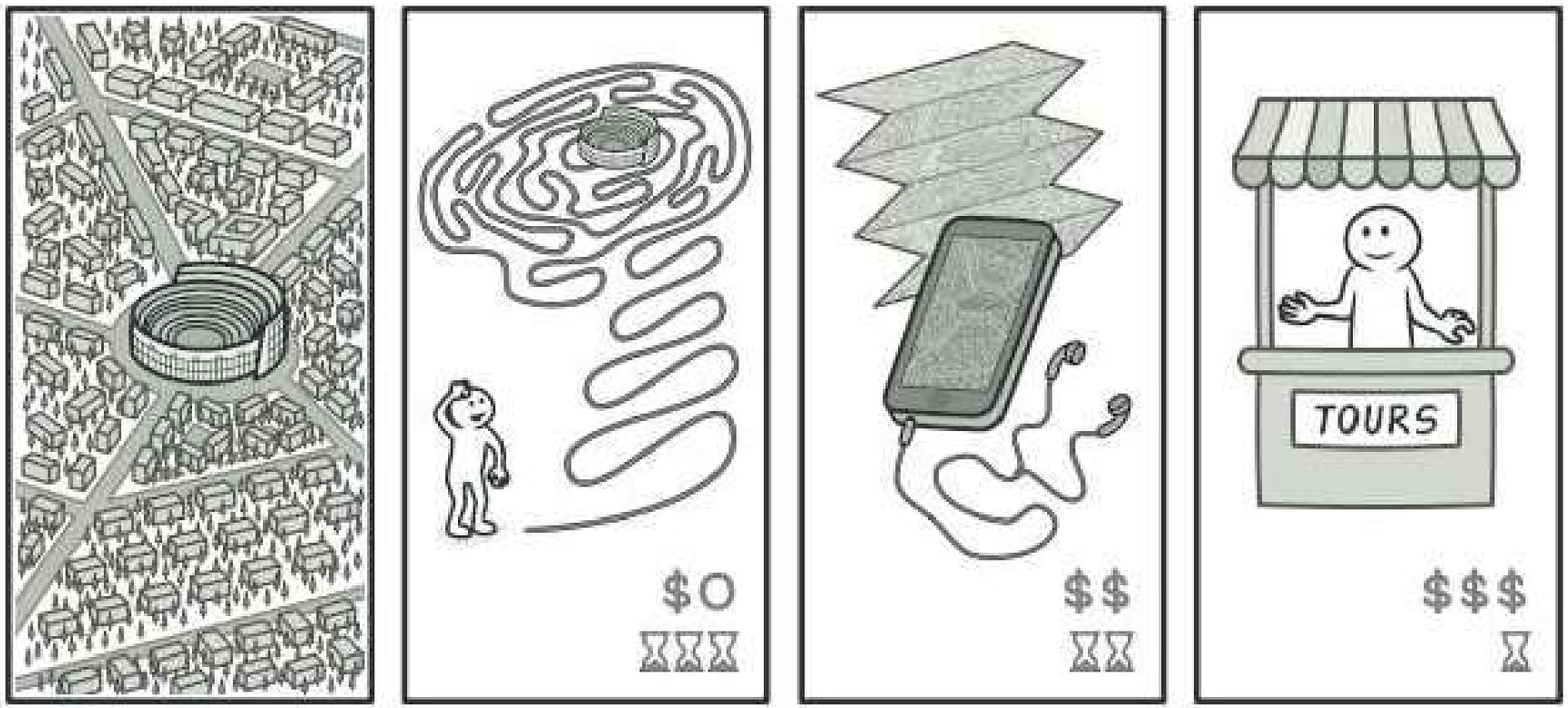
The same collection can be traversed in several different ways.

- 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).

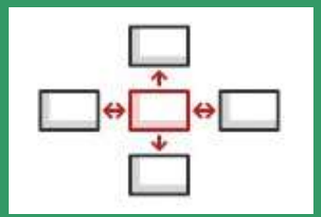
2. Use the pattern to reduce duplication of the traversal code across your app.

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

Introduction

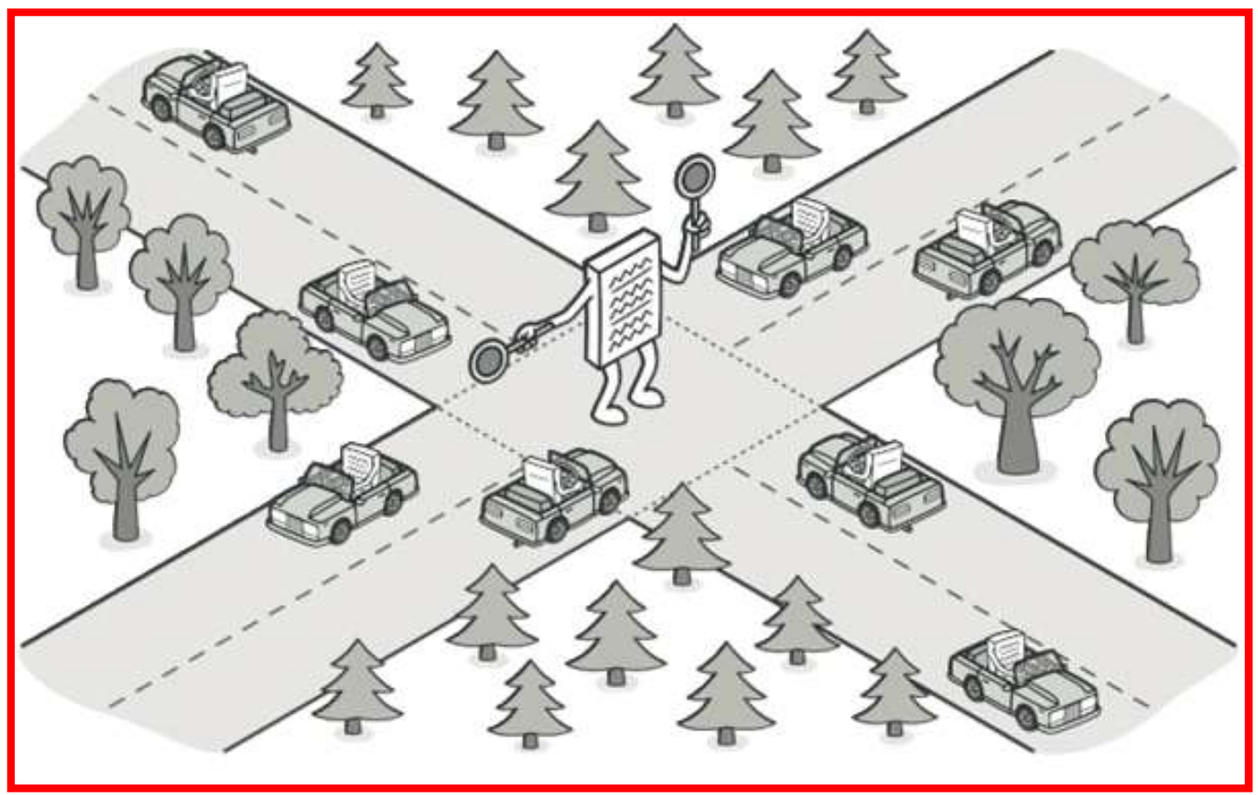
- ***Mediator*** *is a behavioral design pattern that reduces coupling between components of a program by making them communicate indirectly, through a special mediator object.*

- *The Mediator makes it easy to modify, extend and reuse individual components because they're no longer dependent on the dozens of other classes.*

- The mediator design pattern is used to provide a centralized communication medium between different objects in a system. The mediator pattern focuses on providing a mediator between objects for communication and implementing loose-coupling between objects. The mediator works as a router between objects, and it can have its own logic to provide a way of communication.

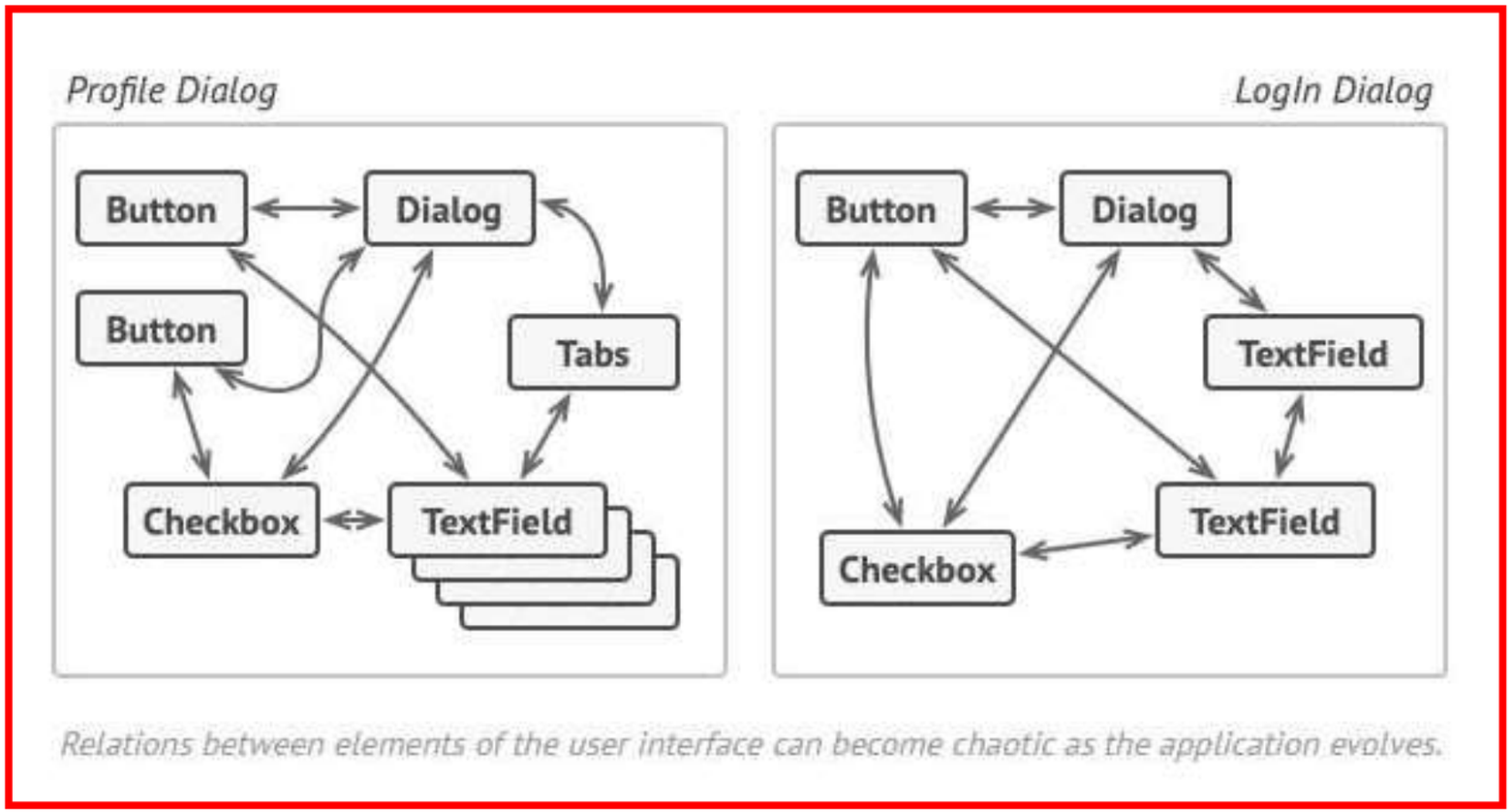
Intent

- **Mediator** is a behavioral design pattern that 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.

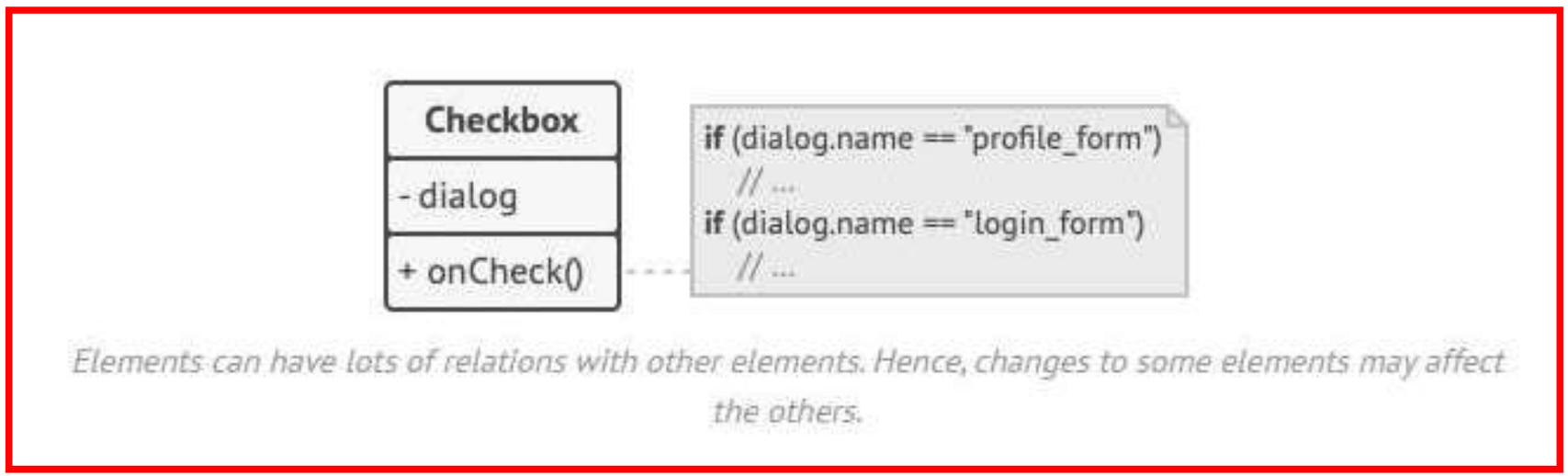


Problem

- Say you have a dialog for creating and editing customer profiles. It consists of various form controls such as text fields, checkboxes, buttons, etc.



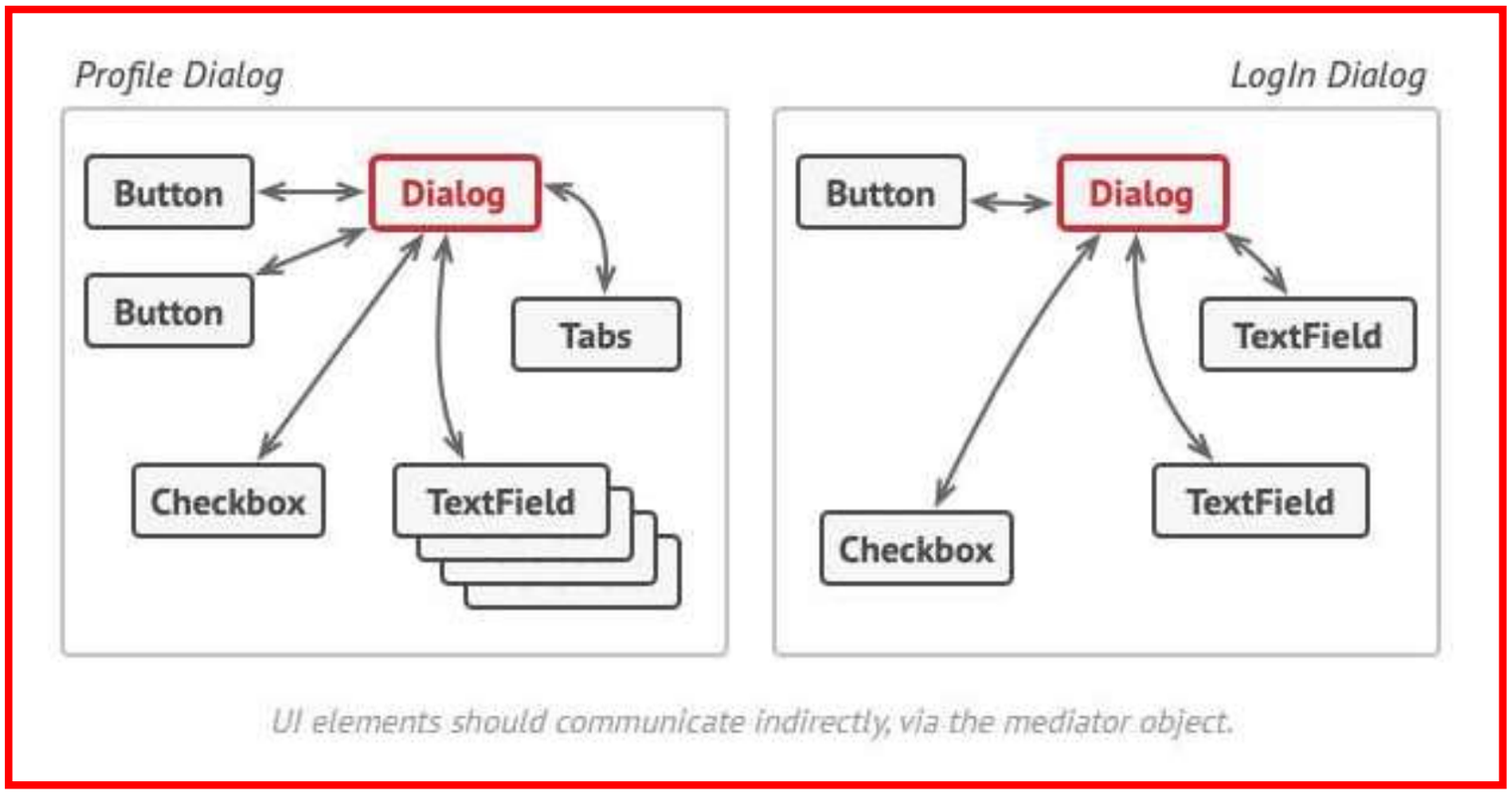
- Some of the form elements may interact with others. For instance, selecting the “I have a dog” checkbox may reveal a hidden text field for entering the dog’s name. Another example is the submit button that has to validate values of all fields before saving the data.



Solution

- The Mediator pattern suggests that you should cease all direct communication between the components which you want to make independent of each other. Instead, these components must collaborate indirectly, by calling a special mediator object that redirects the calls to appropriate components. As a result, the components depend only on a single mediator class instead of being coupled to dozens of their colleagues.
- In our example with the profile editing form, the dialog class itself may act as the mediator. Most likely, the dialog class is already aware of all of its sub-elements, so you won't even need to introduce new dependencies into this class.

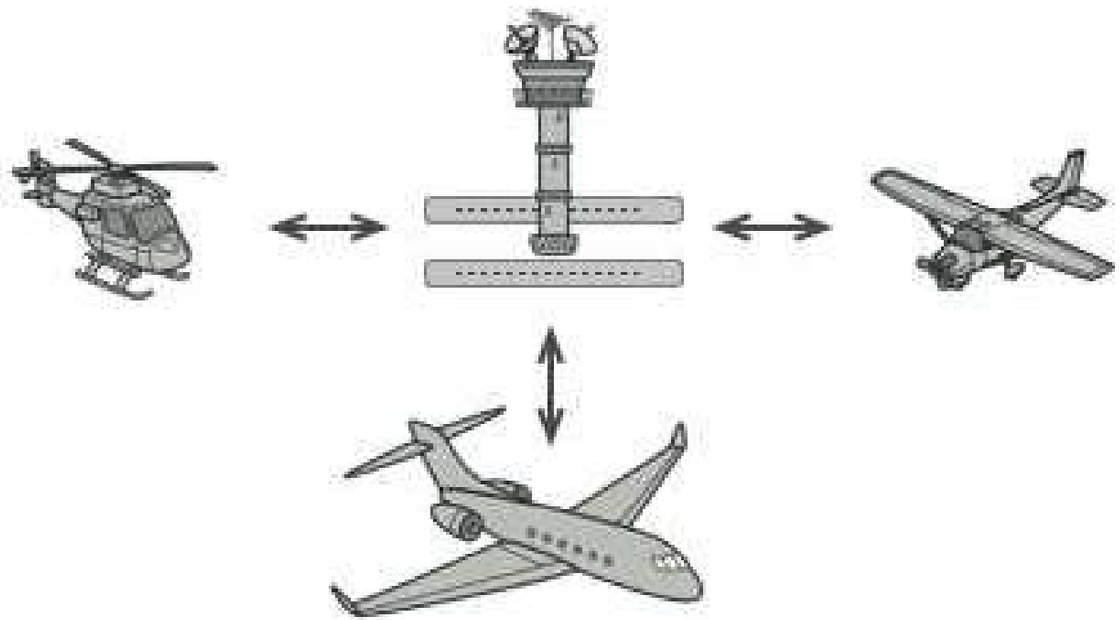
Cont …


Profile Dialog
Button
Dialog
Button
Tabs
Checkbox
TextField
LogIn Dialog
Button
Dialog
TextField
Checkbox
TextField
UI elements should communicate indirectly, via the mediator object.

Cont …

- The most significant change happens to the actual form elements. Let's consider the submit button. Previously, each time a user clicked the button, it had to validate the values of all individual form elements.
- Now its single job is to notify the dialog about the click. Upon receiving this notification, the dialog itself performs the validations or passes the task to the individual elements. Thus, instead of being tied to a dozen form elements, the button is only dependent on the dialog class.

Real-World Analogy



Aircraft pilots don't talk to each other directly when deciding who gets to land their plane next. All communication goes through the control tower.

Complexity: ★★☆

Popularity: ★★☆

Usage examples: The most popular usage of the Mediator pattern in Java code is facilitating communications between GUI components of an app. The synonym of the Mediator is the Controller part of MVC pattern.

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

- **`java.util.Timer`** (all `scheduleXXX()` methods)
- **`java.util.concurrent.Executor#execute()`**
- **`java.util.concurrent.ExecutorService`** (`invokeXXX()` and `submit()` methods)
- **`java.util.concurrent.ScheduledExecutorService`** (all `scheduleXXX()` methods)
- **`java.lang.reflect.Method#invoke()`**

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 Mediator pattern when it's hard to change some of the classes because they are tightly coupled to a bunch of other classes.

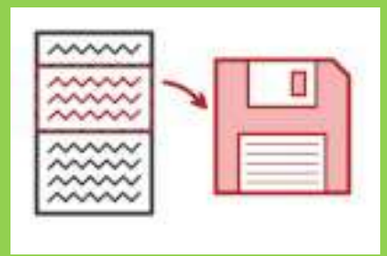
2. Use the pattern when you can't reuse a component in a different program because it's too dependent on other components.

3. Use the Mediator when you find yourself creating tons of component subclasses just to reuse some basic behavior in various contexts.

Pros and Cons

- ✓ *Single Responsibility Principle. You can extract the communications between various components into a single place, making it easier to comprehend and maintain.*
- ✓ *Open/Closed Principle. You can introduce new mediators without having to change the actual components.*
- ✓ *You can reduce coupling between various components of a program.*
- ✓ *You can reuse individual components more easily.*

- ✗ Over time a mediator can evolve into a **God Object**.



Memento Pattern

Introduction

- ***Memento*** *is a behavioral design pattern that allows making snapshots of an object's state and restoring it in future.*

- *The memento design pattern is used when we want to save the state of an object so that we can restore it later on. This pattern is used to implement this in such a way that the saved state data of the object is not accessible outside of the Object, this protects the integrity of saved state data.*

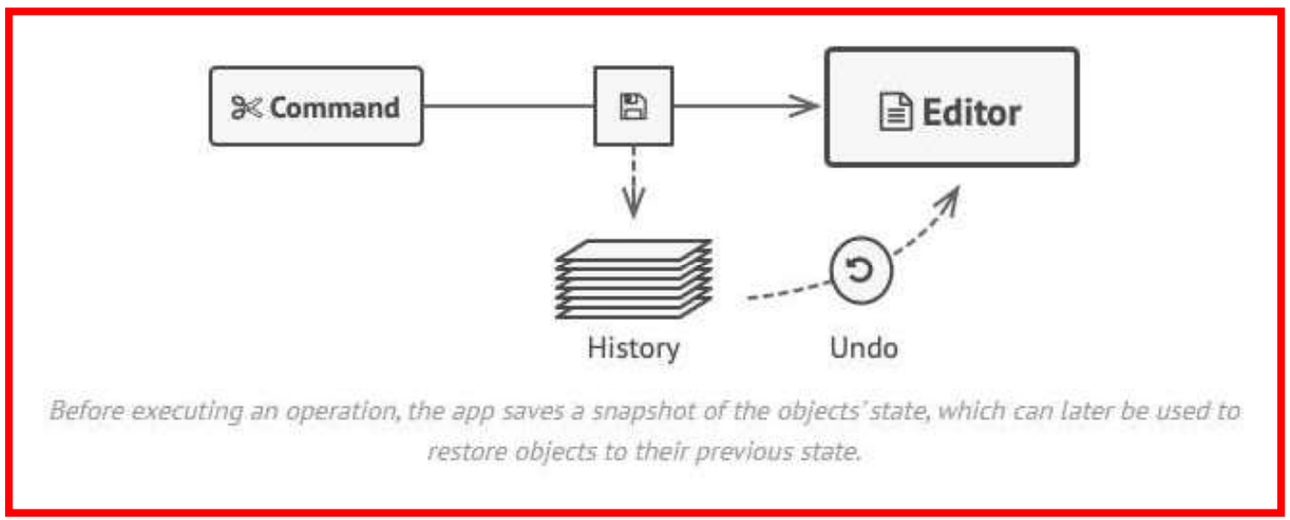
- *Memento pattern is implemented with two Objects – originator and caretaker. The originator is the Object whose state needs to be saved and restored, and it uses an inner class to save the state of Object. The inner class is called "Memento", and it's private so that it can't be accessed from other objects.*

Intent

- **Memento** is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

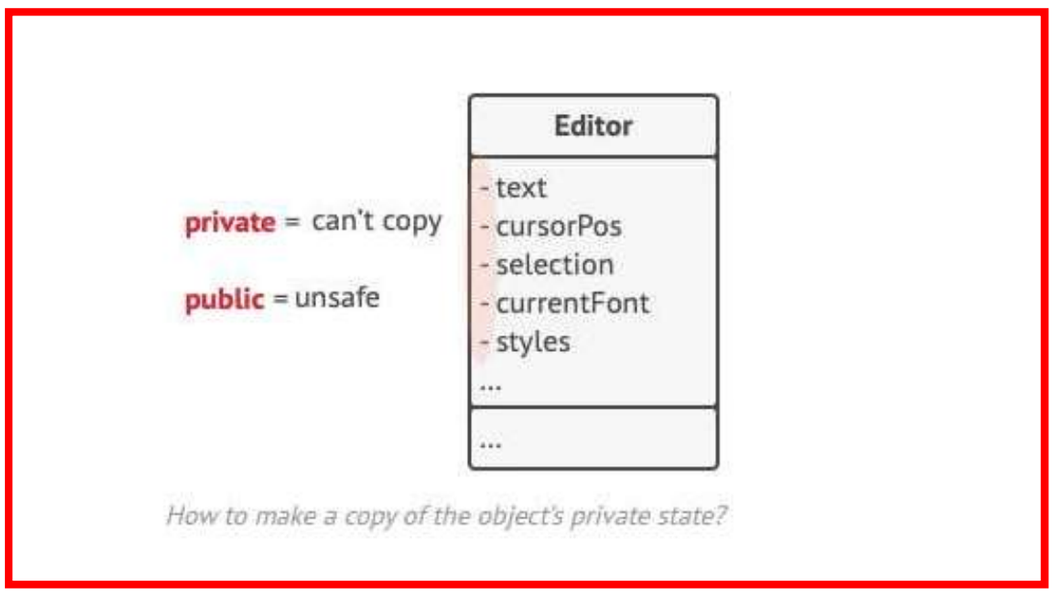
Problem

- Imagine that you're creating a text editor app. In addition to simple text editing, your editor can format text, insert inline images, etc.
- At some point, you decided to let users undo any operations carried out on the text. This feature has become so common over the years that nowadays people expect every app to have it. For the implementation, you chose to take the direct approach. Before performing any operation, the app records the state of all objects and saves it in some storage. Later, when a user decides to revert an action, the app fetches the latest snapshot from the history and uses it to restore the state of all objects.



Before executing an operation, the app saves a snapshot of the objects' state, which can later be used to restore objects to their previous state.

- Let’s think about those state snapshots. How exactly would you produce one? You’d probably need to go over all the fields in an object and copy their values into storage. However, this would only work if the object had quite relaxed access restrictions to its contents. Unfortunately, most real objects won’t let others peek inside them that easily, hiding all significant data in private fields.



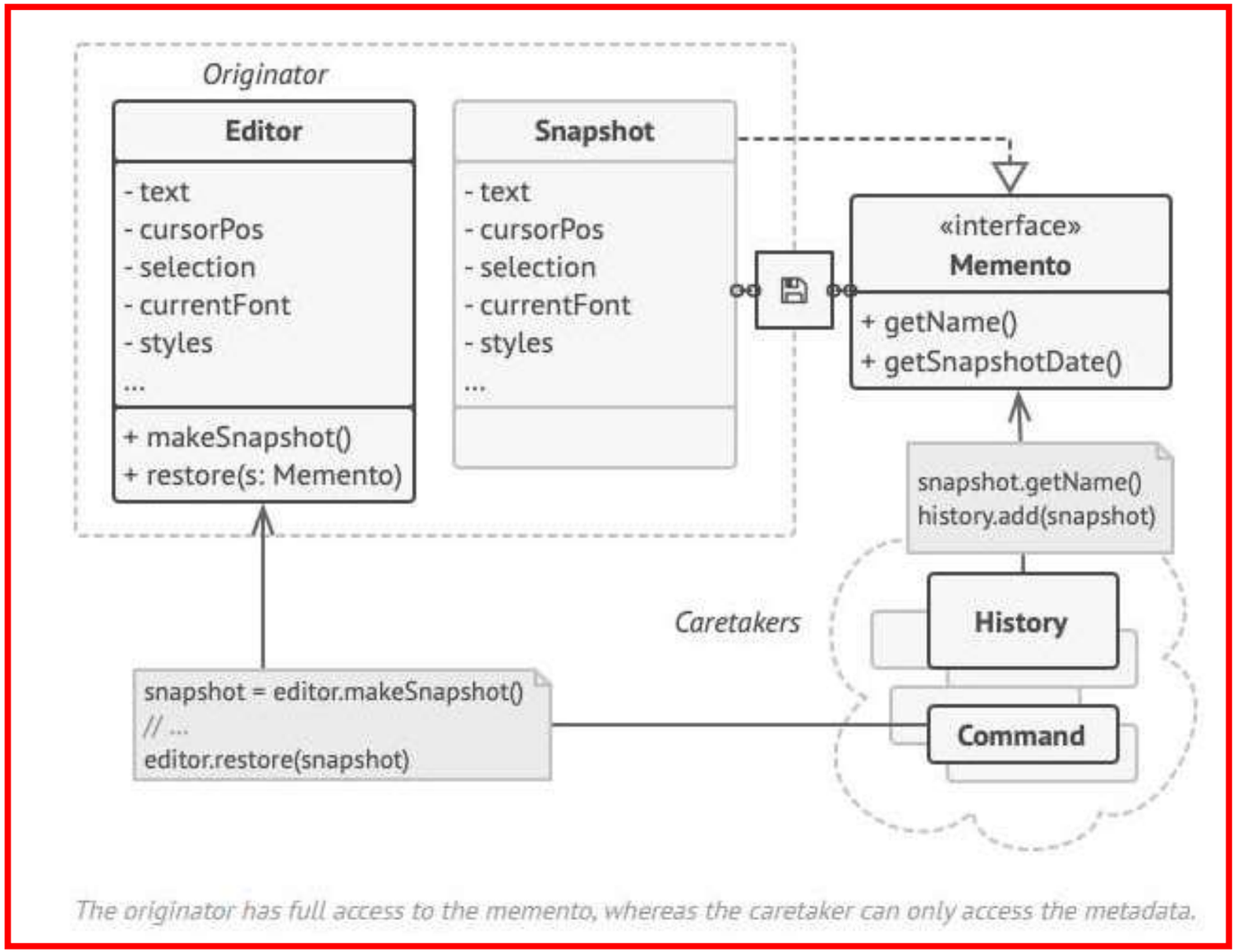
How to make a copy of the object's private state?

- All problems that we've just experienced are caused by broken encapsulation. Some objects try to do more than they are supposed to. To collect the data required to perform some action, they invade the private space of other objects instead of letting these objects perform the actual action.

- The Memento pattern delegates creating the state snapshots to the actual owner of that state, the *originator* object. Hence, instead of other objects trying to copy the editor's state from the "outside," the editor class itself can make the snapshot since it has full access to its own state.

- 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.


Originator
Editor
- text
- cursorPos
- selection
- currentFont
- styles
...
+ makeSnapshot()
+ restore(s: Memento)
Snapshot
- text
- cursorPos
- selection
- currentFont
- styles
...
«interface»
Memento
+ getName()
+ getSnapshotDate()
snapshot.getName()
history.add(snapshot)
Caretakers
History
Command
snapshot = editor.makeSnapshot()
// ...
editor.restore(snapshot)
The originator has full access to the memento, whereas the caretaker can only access the metadata.

Complexity: ★★★

Popularity: ★☆☆

Usage examples: The Memento's principle can be achieved using serialization, which is quite common in Java. While it's not the only and the most efficient way to make snapshots of an object's state, it still allows storing state backups while protecting the originator's structure from other objects.

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

- All `java.io.Serializable` implementations can simulate the Memento.
- All `javax.faces.component.StateHolder` implementations.

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 Memento pattern when you want to produce snapshots of the object's state to be able to restore a previous state of the object.

2. Use the pattern when direct access to the object's fields/getters/setters violates its encapsulation.

Pros and Cons

- ✓ *You can produce snapshots of the object's state without violating its encapsulation.*
- ✓ *You can simplify the originator's code by letting the caretaker maintain the history of the originator's state*.

- ✘ The app might consume lots of RAM if clients create mementos too often.
- ✘ Caretakers should track the originator's lifecycle to be able to destroy obsolete mementos.

Observer Pattern

Introduction

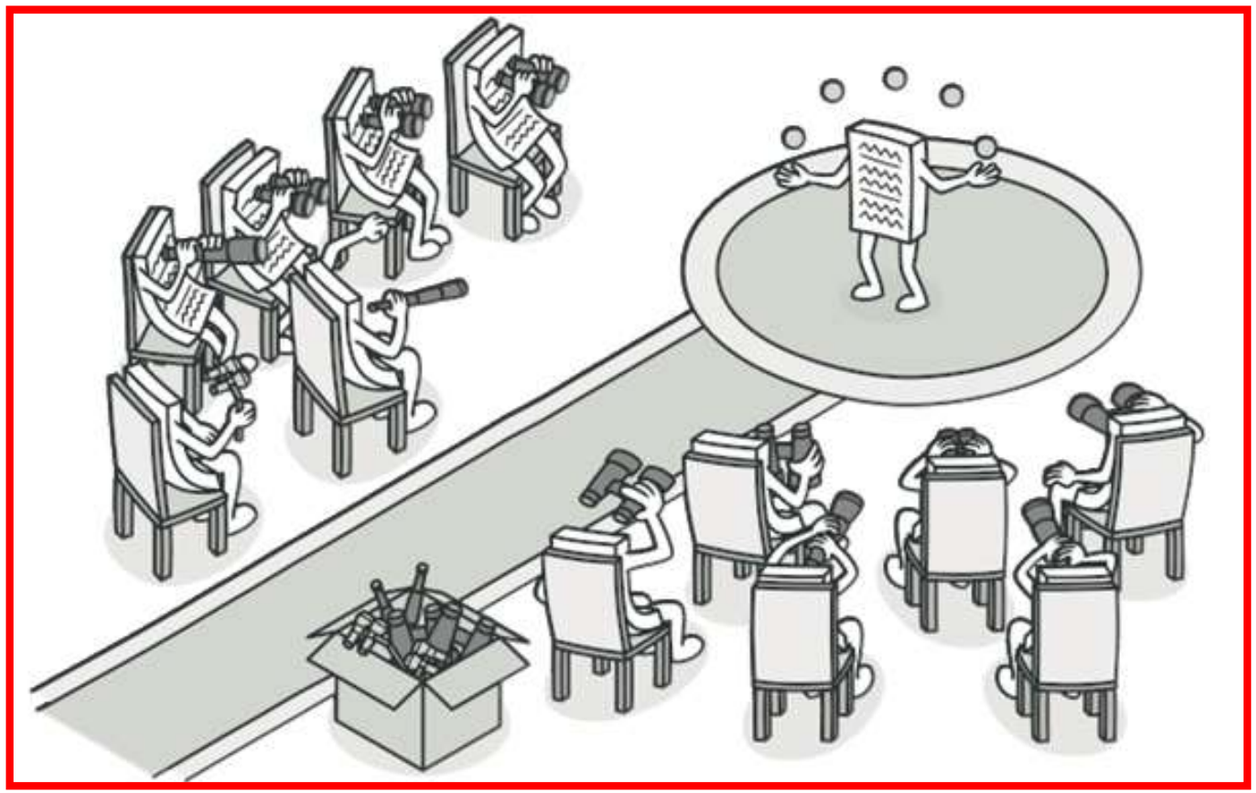
- ***Observer*** *is a behavioral design pattern that allows some objects to notify other objects about changes in their state.*

- *The Observer pattern provides a way to subscribe and unsubscribe to and from these events for any object that implements a subscriber interface.*

- *An observer design pattern is useful when you are interested in the state of an Object and want to get notified whenever there is any change. In the observer pattern, the Object that watches the state of another Object is called observer, and the Object that is being watched is called subject.*

Intent

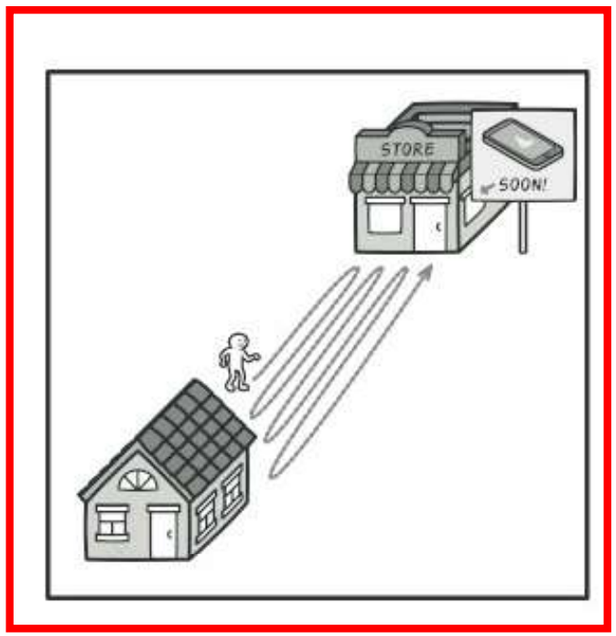
- **Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



Problem

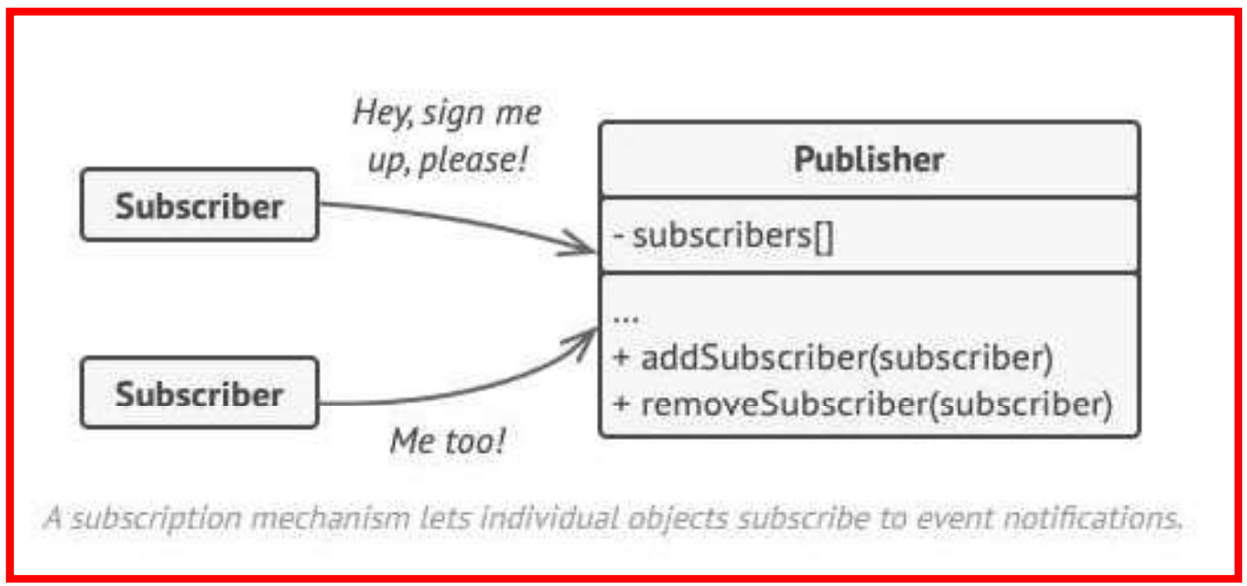
- Imagine that you have two types of objects: a Customer and a Store. The customer is very interested in a particular brand of product (say, it's a new model of the iPhone) which should become available in the store very soon.

- The customer could visit the store every day and check product availability. But while the product is still en route, most of these trips would be pointless.

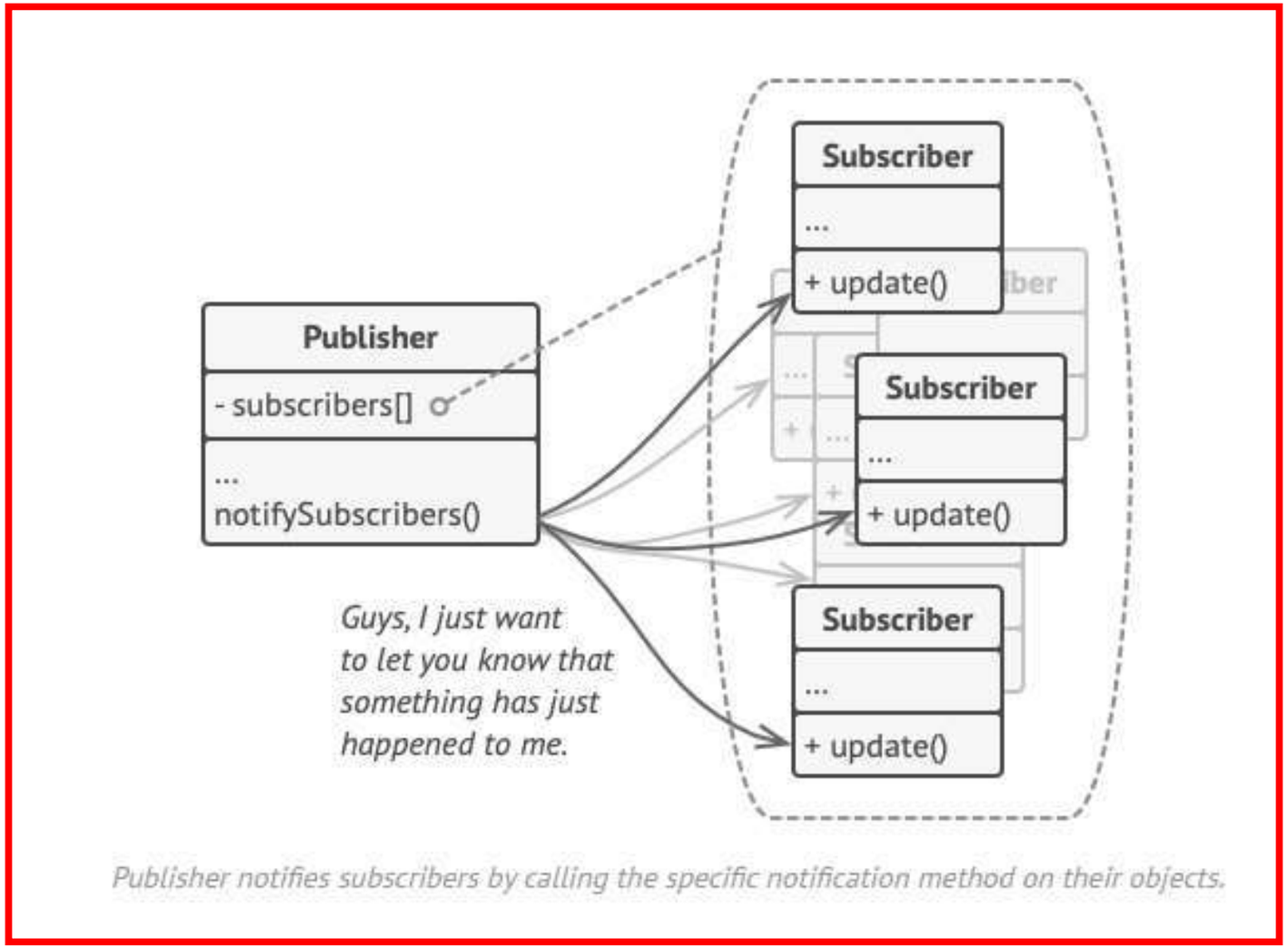


- On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it'd upset other customers who aren't interested in new products.
- It looks like we've got a conflict. Either the customer wastes time checking product availability or the store wastes resources notifying the wrong customers.

- The object that has some interesting state is often called ***subject***, but since it's also going to notify other objects about the changes to its state, we'll call it ***publisher***. All other objects that want to track changes to the publisher's state are called ***subscribers***.

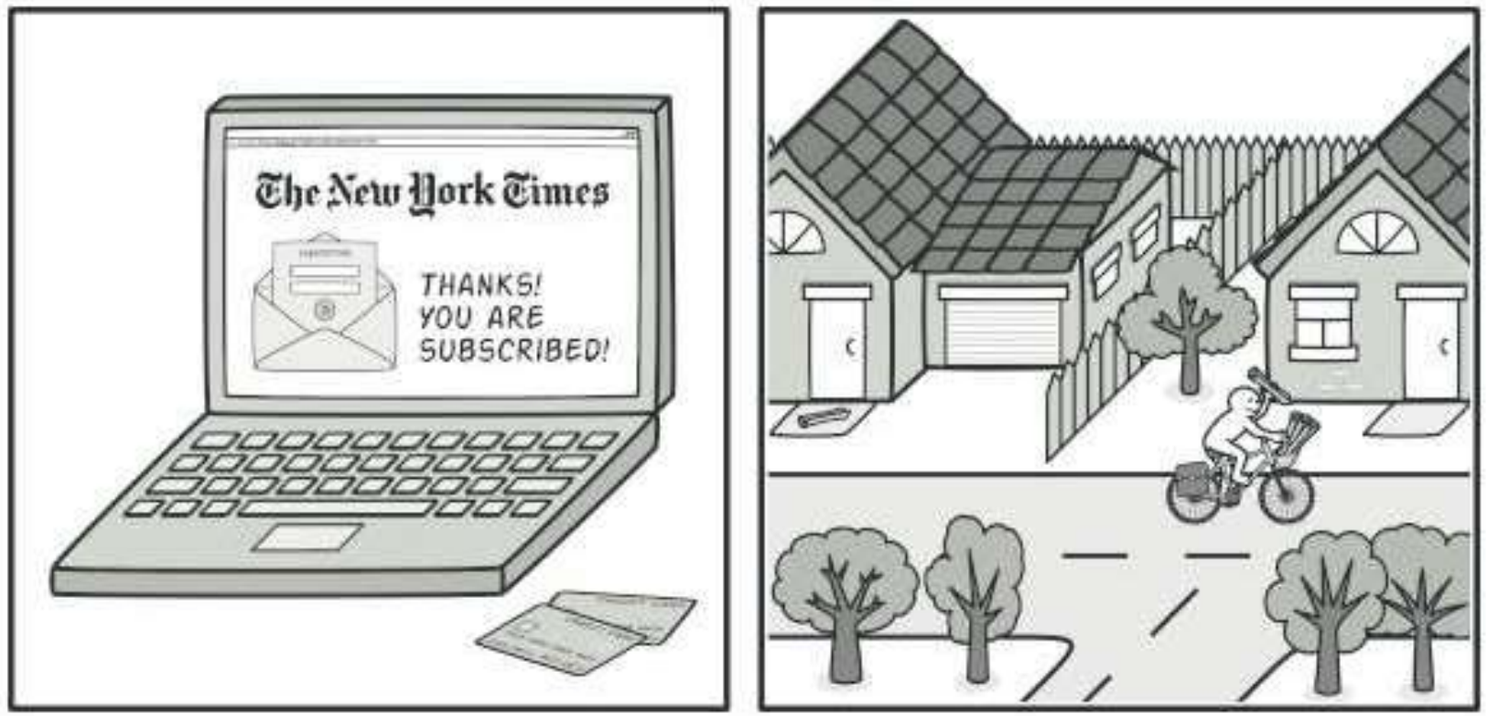


- Now, whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects.
- That's why it's crucial that all subscribers implement the same interface and that the publisher communicates with them only via that interface. This interface should declare the notification method along with a set of parameters that the publisher can use to pass some contextual data along with the notification.



Publisher notifies subscribers by calling the specific notification method on their objects.

Real-World Analogy



Magazine and newspaper subscriptions.

Complexity: ★★☆

Popularity: ★★★

Usage examples: The Observer pattern is pretty common in Java code, especially in the GUI components. It provides a way to react to events happening in other objects without coupling to their classes.

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

- `java.util.Observer` / `java.util.Observable` (rarely used in real world)
- All implementations of `java.util.EventListener` (practically all over Swing components)
- `javax.servlet.http.HttpSessionBindingListener`
- `javax.servlet.http.HttpSessionAttributeListener`
- `javax.faces.event.PhaseListener`

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

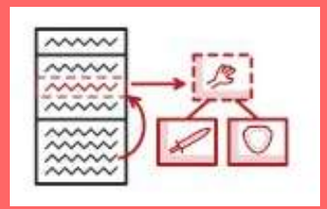
1. Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.

2. Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases.

Pros and Cons

- ✓ *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.*

- ✘ Subscribers are notified in random order.



Strategy Pattern

Introduction

- **Strategy** is a behavioral design pattern that turns a set of behaviors into objects and makes them interchangeable inside original context object.

- Strategy pattern is used when we have multiple algorithms for a specific task, and the client decides the actual implementation be used at runtime. A strategy pattern is also known as a policy pattern. We define multiple algorithms and let client applications pass the algorithm to be used as a parameter.

- One of the best examples of this pattern is the **Collections.sort**() method that takes the Comparator parameter. Based on the different implementations of comparator interfaces, the objects are getting sorted in different ways.

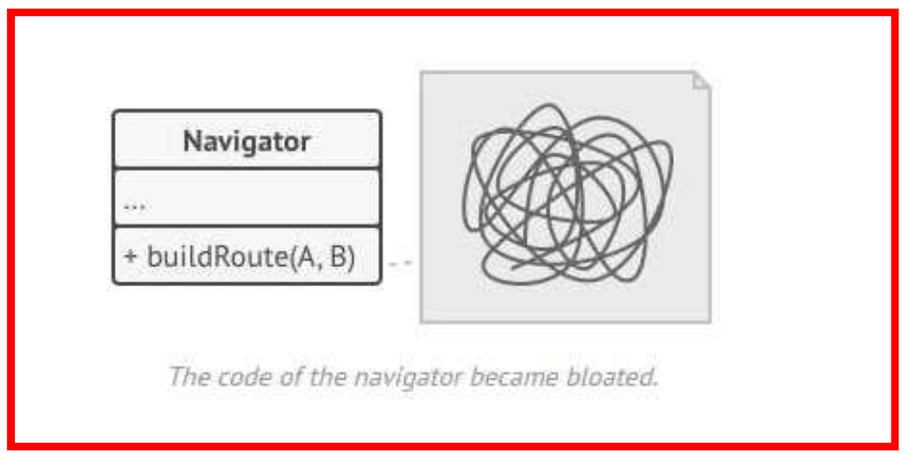
Intent

- **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.

Problem

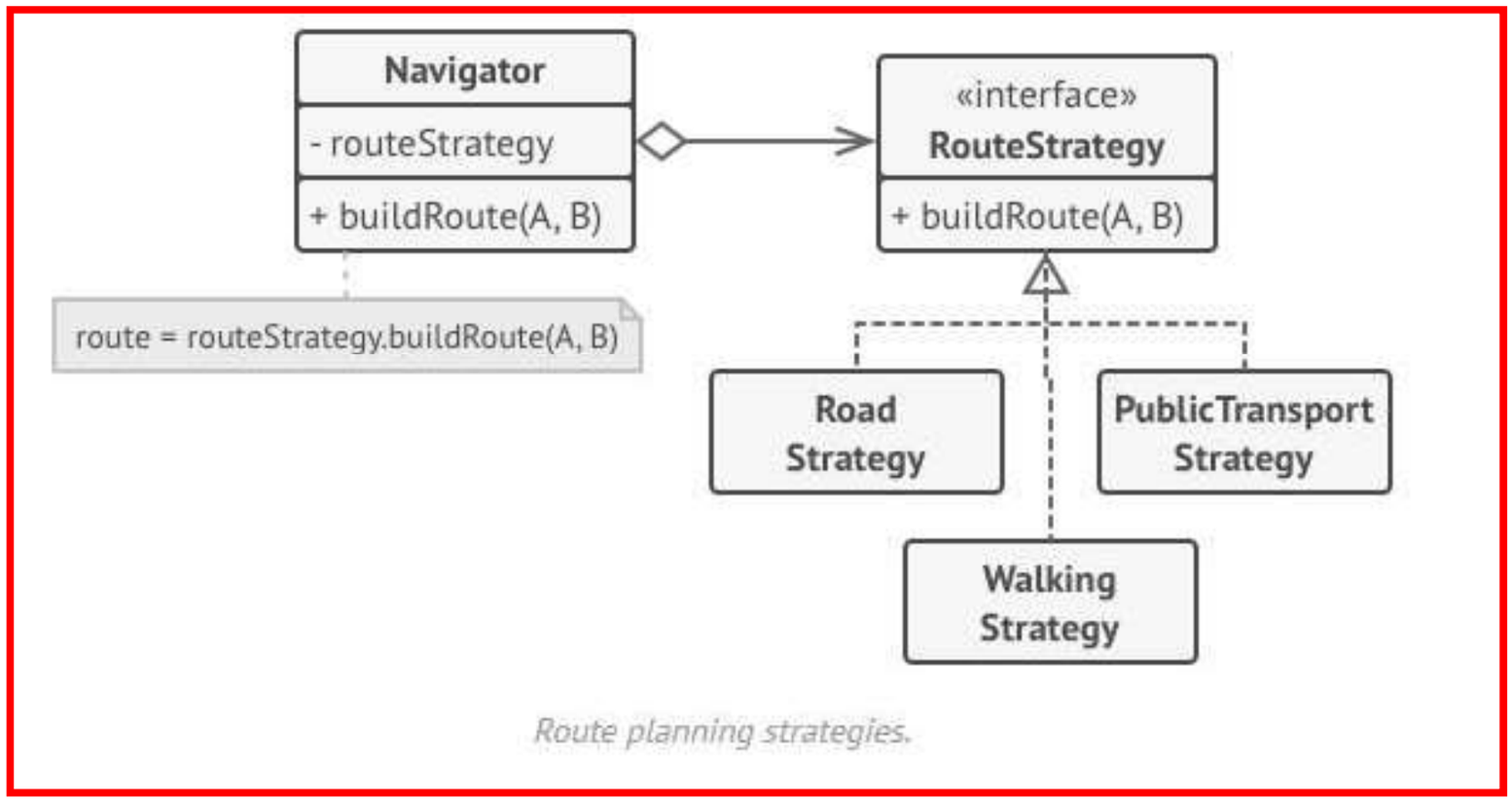
- One day you decided to create a navigation app for casual travelers. The app was centered around a beautiful map which helped users quickly orient themselves in any city.
- One of the most requested features for the app was automatic route planning. A user should be able to enter an address and see the fastest route to that destination displayed on the map.
- The first version of the app could only build the routes over roads. People who traveled by car were bursting with joy. But apparently, not everybody likes to drive on their vacation. So with the next update, you added an option to build walking routes. Right after that, you added another option to let people use public transport in their routes.
- However, that was only the beginning. Later you planned to add route building for cyclists. And even later, another option for building routes through all of a city's tourist attractions.

- While from a business perspective the app was a success, the technical part caused you many headaches. Each time you added a new routing algorithm, the main class of the navigator doubled in size. At some point, the beast became too hard to maintain.
- Any change to one of the algorithms, whether it was a simple bug fix or a slight adjustment of the street score, affected the whole class, increasing the chance of creating an error in already-working code.



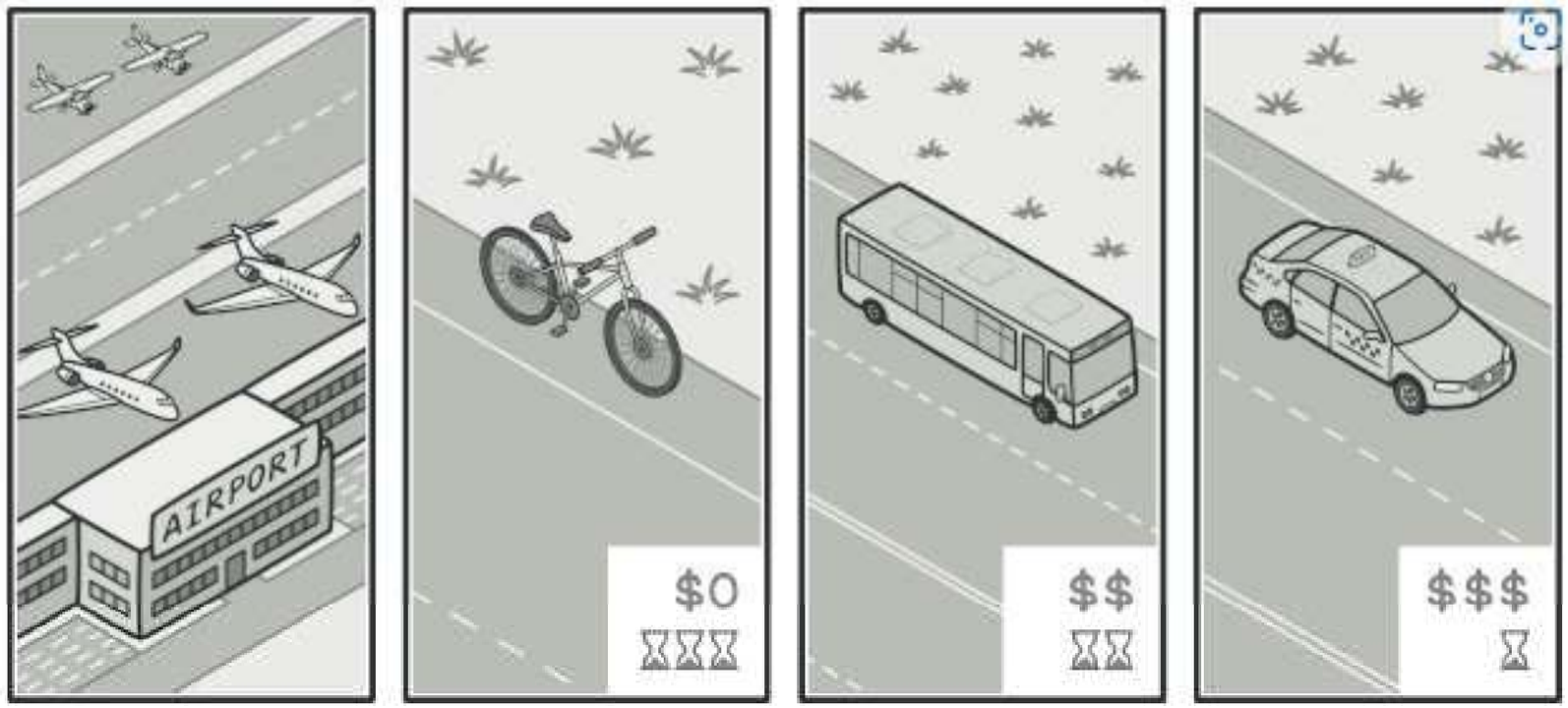
The code of the navigator became bloated.

- The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*.
- The original class, called *context*, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.
- The context isn't responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context. In fact, the context doesn't know much about strategies. It works with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy.



Route planning strategies.

Real-World Analogy



Various strategies for getting to the airport.

Complexity: ★☆☆

Popularity: ★★★

Usage examples: The Strategy pattern is very common in Java code. It's often used in various frameworks to provide users a way to change the behavior of a class without extending it.

Java 8 brought the support of lambda functions, which can serve as simpler alternatives to the Strategy pattern.

Here some examples of Strategy in core Java libraries:

- **`java.util.Comparator#compare()`** called from `Collections#sort()`.
- **`javax.servlet.http.HttpServlet`**: `service()` method, plus all of the `doXXX()` methods that accept `HttpServletRequest` and `HttpServletResponse` objects as arguments.
- **`javax.servlet.Filter#doFilter()`**

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 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.

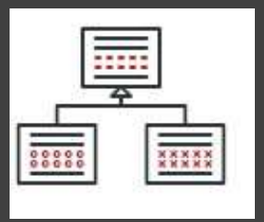
2. Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior.

3. Use the pattern to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.

Pros and Cons

- ✓ *You can swap algorithms used inside an object at runtime.*
- ✓ *You can isolate the implementation details of an algorithm from the code that uses it.*
- ✓ *You can replace inheritance with composition.*
- ✓ *Open/Closed Principle. You can introduce new strategies without having to change the context.*

- ✗ Clients must be aware of the differences between strategies to be able to select a proper one.



Template Pattern

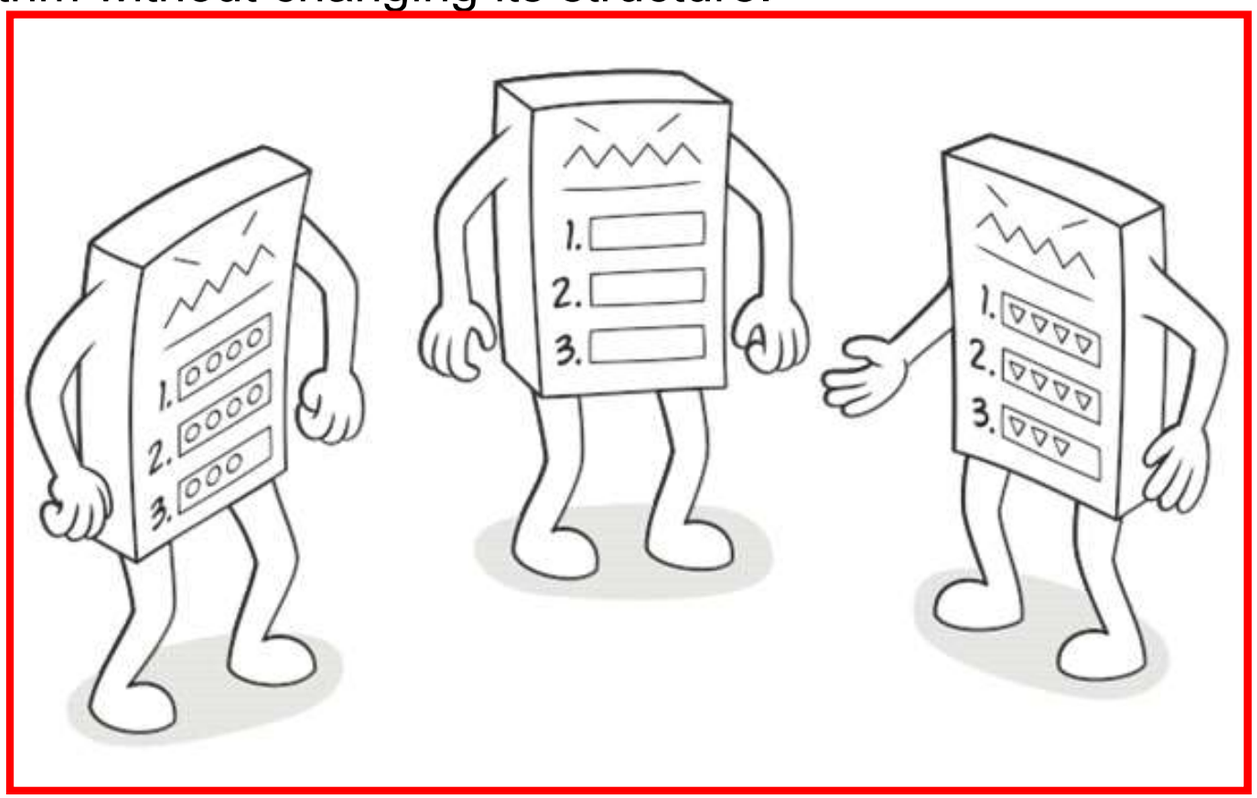
Introduction

- **Template Method** is a behavioral design pattern that allows you to defines a skeleton of an algorithm in a base class and let subclasses override the steps without changing the overall algorithm's structure.

- The template method pattern is a behavioral design pattern and is used to create a method stub and to defer some of the steps of implementation to the subclasses. The template method defines the steps to execute an algorithm, and it can provide a default implementation that might be common for all or some of the subclasses.

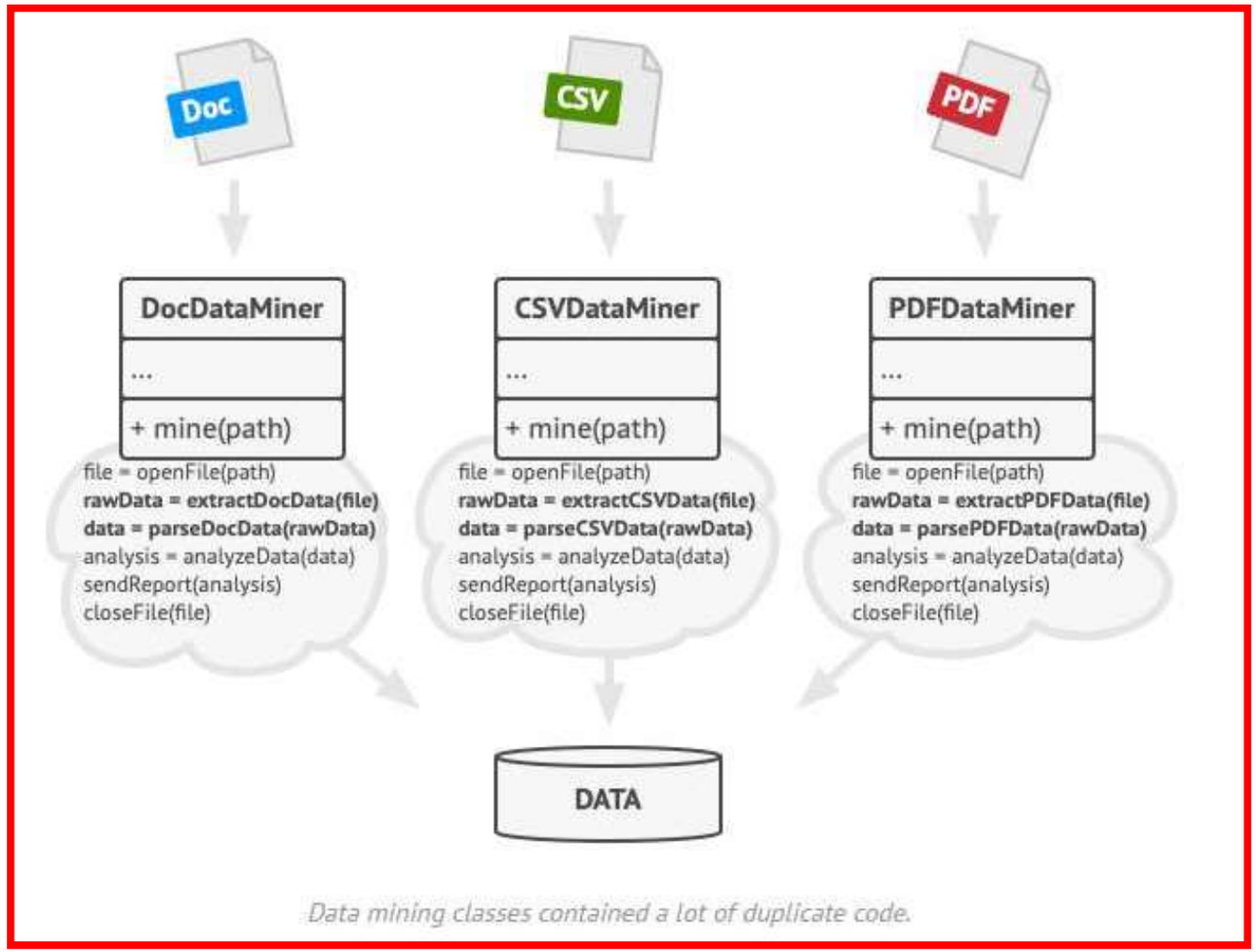
Intent

- **Template Method** is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.



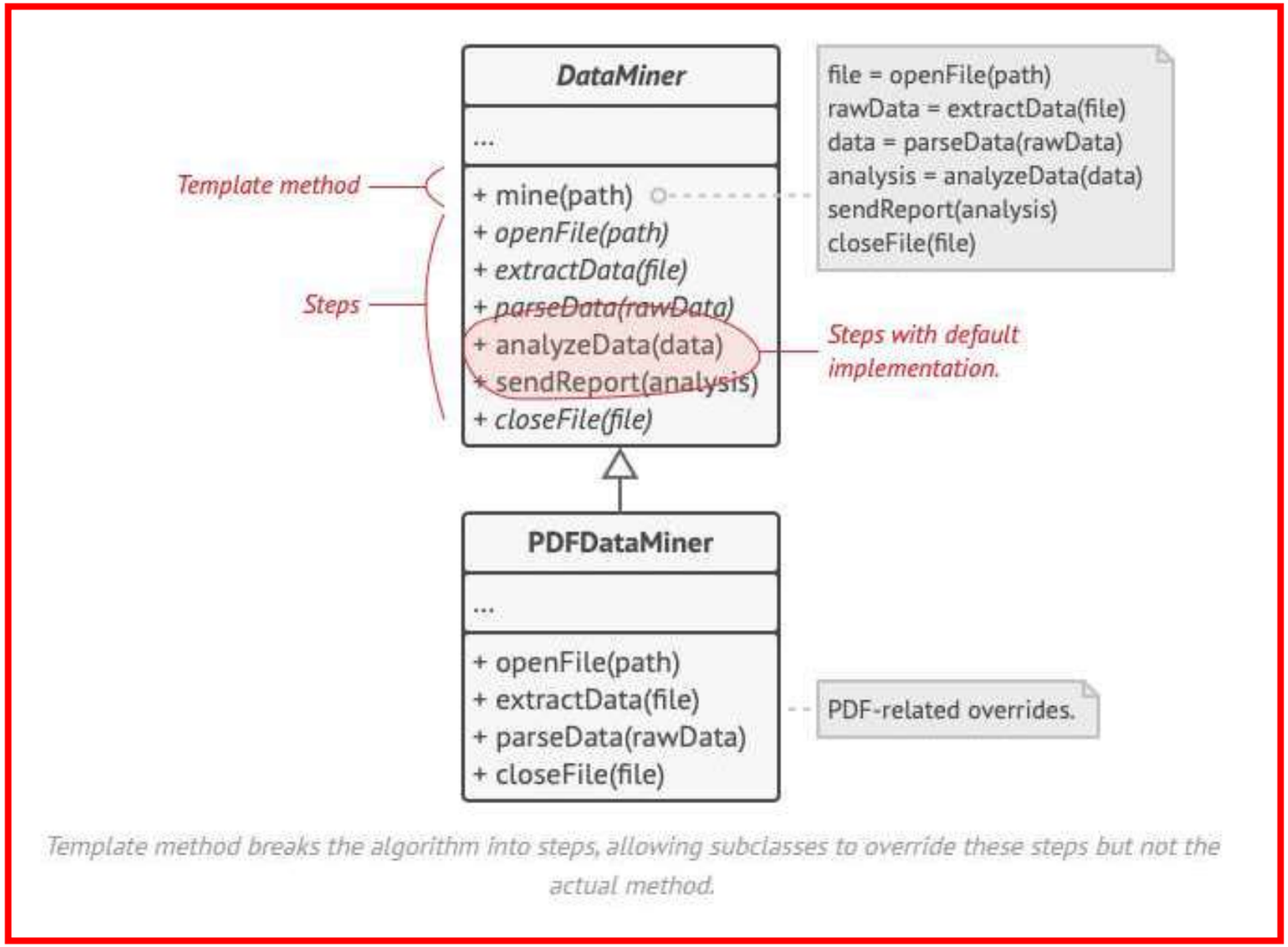
Problem

- Imagine that you're creating a data mining application that analyzes corporate documents. Users feed the app documents in various formats (PDF, DOC, CSV), and it tries to extract meaningful data from these docs in a uniform format.
- The first version of the app could work only with DOC files. In the following version, it was able to support CSV files. A month later, you "taught" it to extract data from PDF files.
- At some point, you noticed that all three classes have a lot of similar code. While the code for dealing with various data formats was entirely different in all classes, the code for data processing and analysis is almost identical. Wouldn't it be great to get rid of the code duplication, leaving the algorithm structure intact?



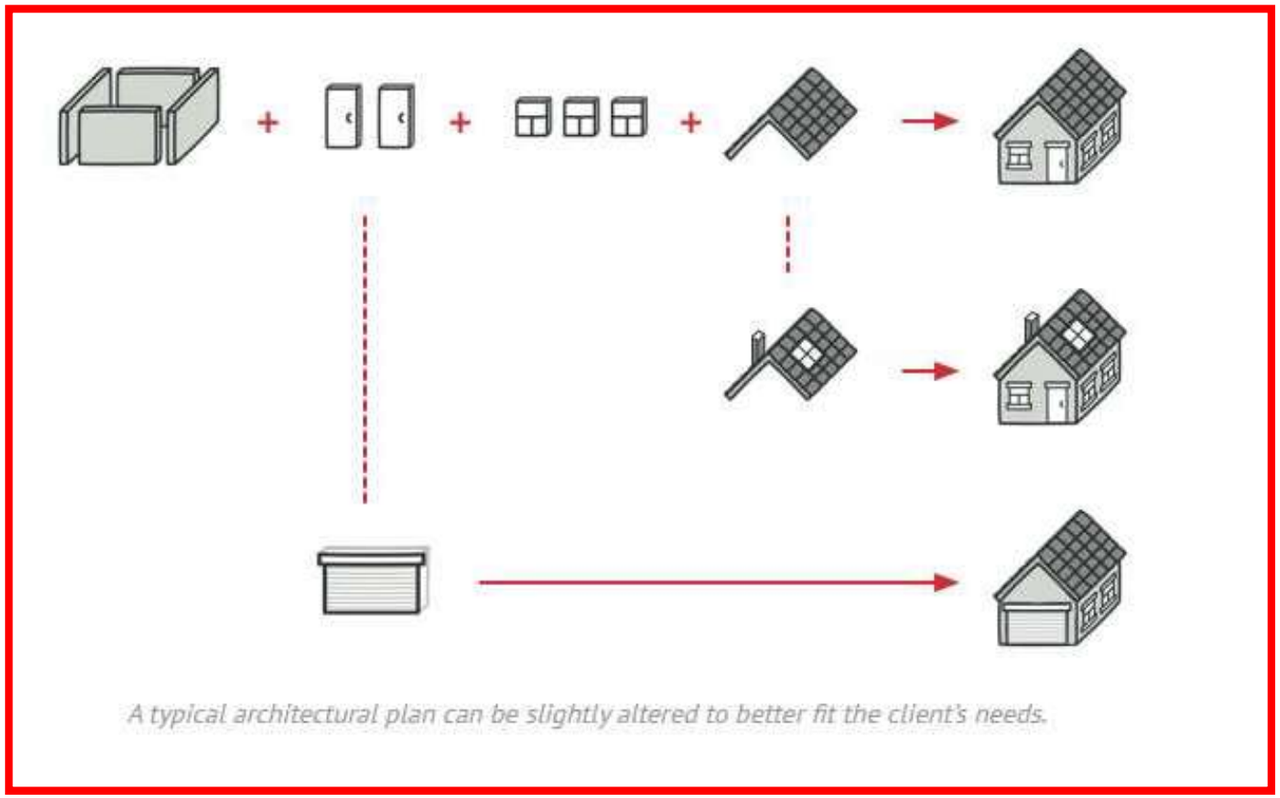
Data mining classes contained a lot of duplicate code.

- The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single template method.
- The steps may either be abstract, or have some default implementation. To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).



Template method breaks the algorithm into steps, allowing subclasses to override these steps but not the actual method.

Real-World Analogy



A typical architectural plan can be slightly altered to better fit the client's needs.

Complexity: ★☆☆

Popularity: ★★☆

Usage examples: The Template Method pattern is quite common in Java frameworks. Developers often use it to provide framework users with a simple means of extending standard functionality using inheritance.

Here are some examples of Template Methods in core Java libraries:

- All non-abstract methods of `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader` and `java.io.Writer`.
- All non-abstract methods of `java.util.AbstractList`, `java.util.AbstractSet` and `java.util.AbstractMap`.
- In `javax.servlet.http.HttpServlet` class, all the `doXXX()` methods send the HTTP 405 "Method Not Allowed" error by default. However, you can override any of those methods to send a different response.

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

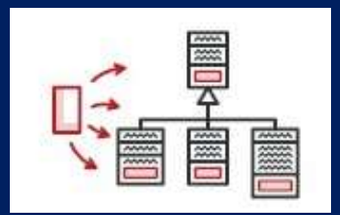
1. Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure.

2. Use the pattern when you have several classes that contain almost identical algorithms with some minor differences. As a result, you might need to modify all classes when the algorithm changes.

Pros and Cons

- ✓ *You can let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm.*
- ✓ *You can pull the duplicate code into a superclass.*

- ✘ Some clients may be limited by the provided skeleton of an algorithm.
- ✘ Template methods tend to be harder to maintain the more steps they have.



Visitor Pattern

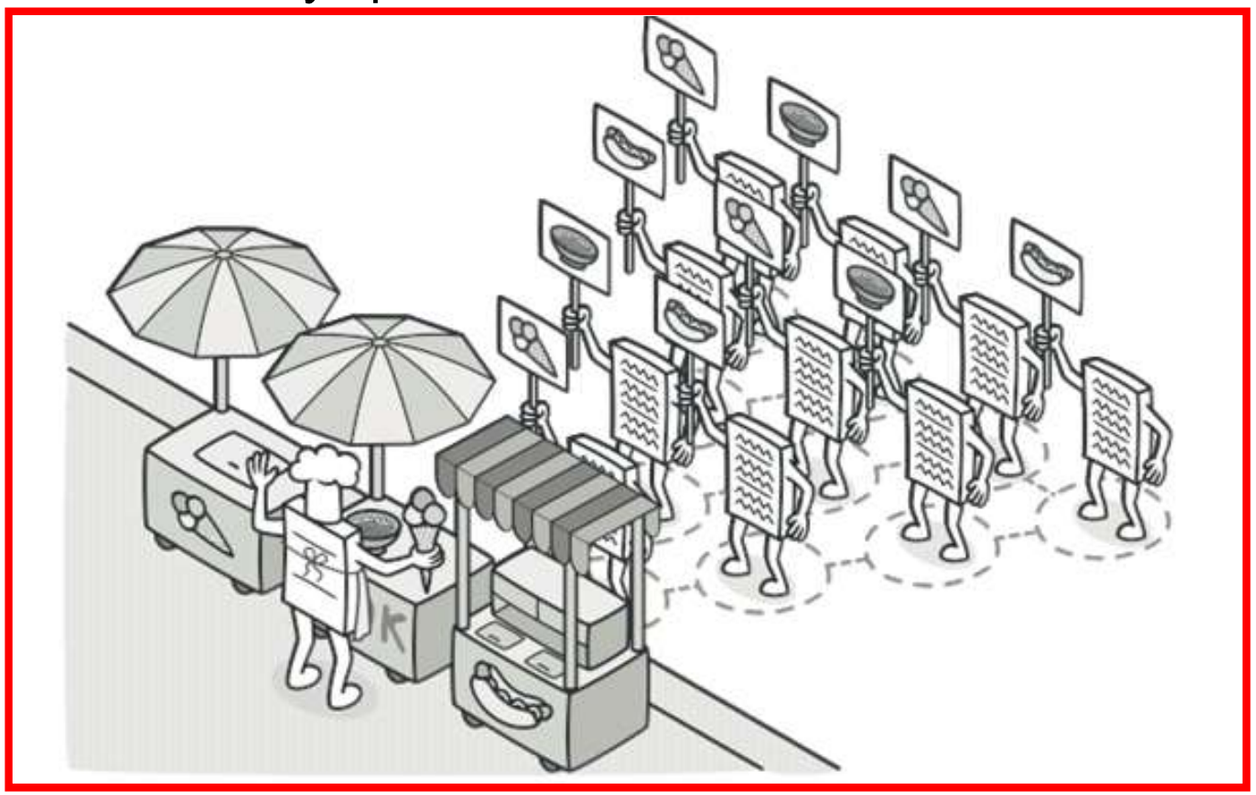
Introduction

- ***Visitor*** *is a behavioral design pattern that allows adding new behaviors to existing class hierarchy without altering any existing code.*

- The visitor pattern is used when we have to perform an operation on a group of similar kinds of objects. With the help of a visitor pattern, we can move the operational logic from the objects to another class.

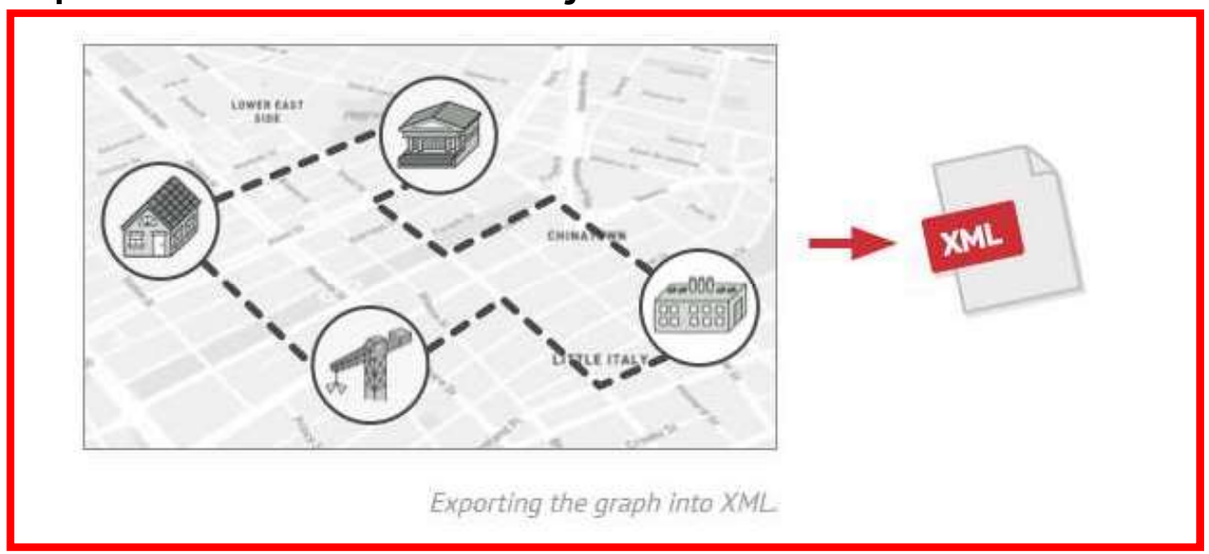
Intent

- **Visitor** is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

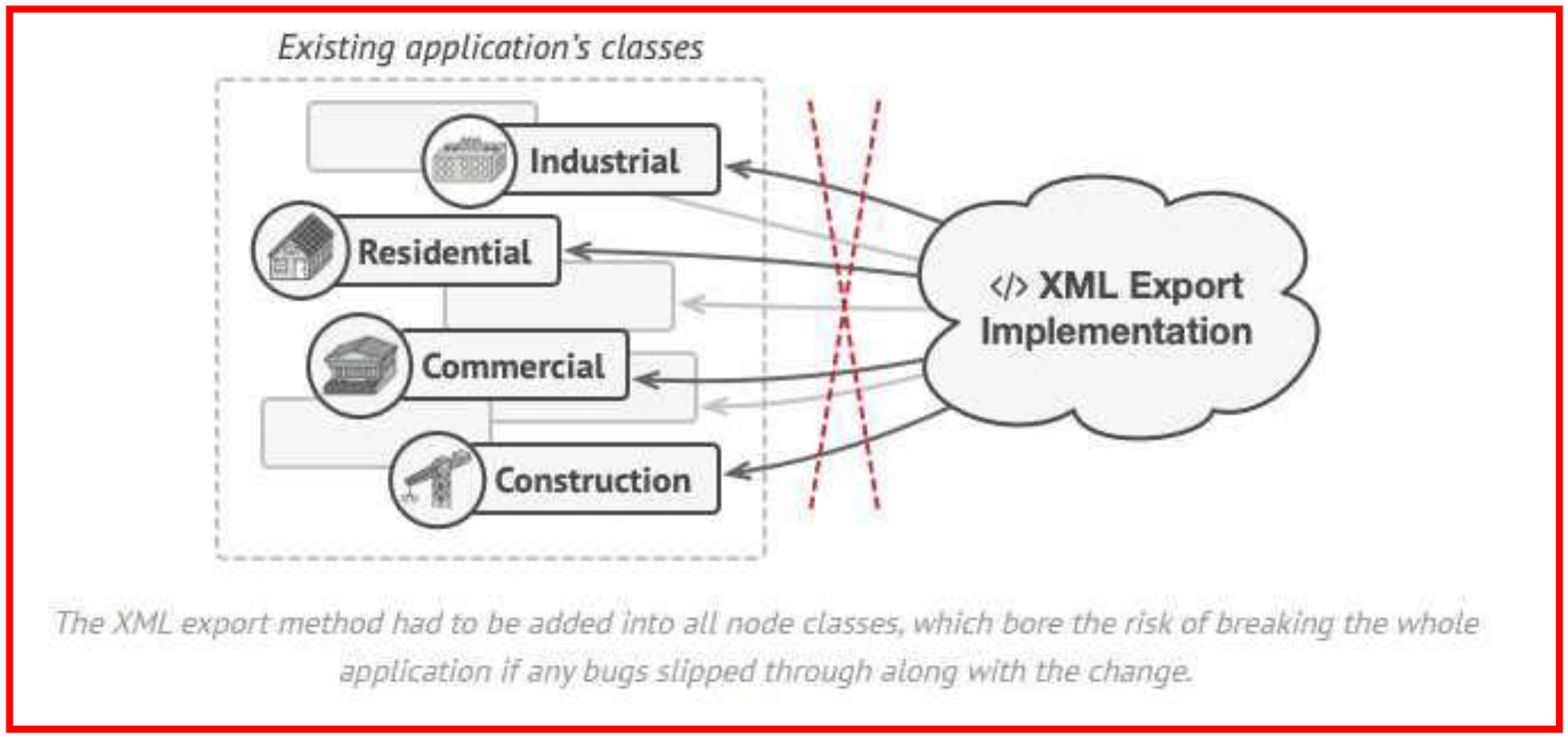


Problem

- Imagine that your team develops an app which works with geographic information structured as one colossal graph.
- Each node of the graph may represent a complex entity such as a city, but also more granular things like industries, sightseeing areas, etc.
- The nodes are connected with others if there's a road between the real objects that they represent. Under the hood, each node type is represented by its own class, while each specific node is an object.



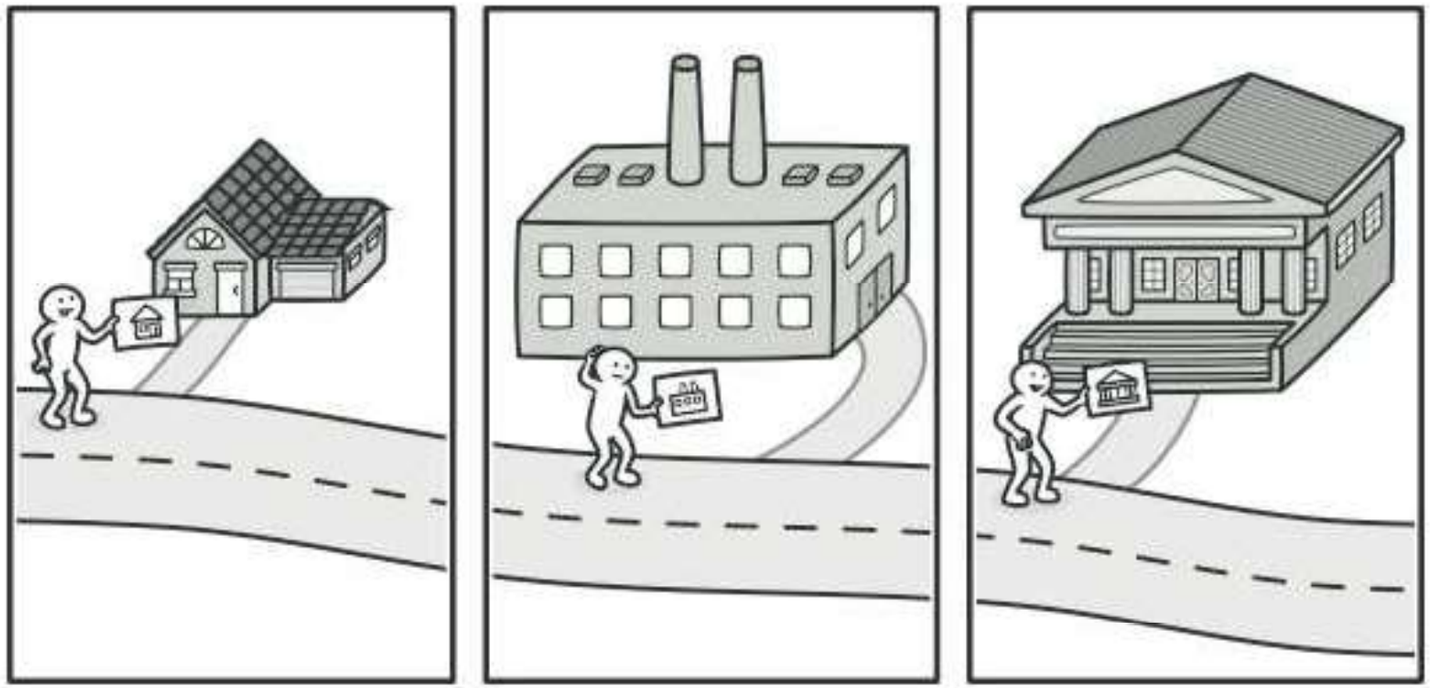
Exporting the graph into XML



The XML export method had to be added into all node classes, which bore the risk of breaking the whole application if any bugs slipped through along with the change.

- The Visitor pattern suggests that you place the new behavior into a separate class called *visitor*, instead of trying to integrate it into existing classes.
- The original object that had to perform the behavior is now passed to one of the visitor's methods as an argument, providing the method access to all necessary data contained within the object.

Real-World Analogy



A good insurance agent is always ready to offer different policies to various types of organizations.

Complexity: ★★★

Popularity: ★☆☆

Usage examples: Visitor isn't a very common pattern because of its complexity and narrow applicability.

Here are some examples of pattern in core Java libraries:

- `javax.lang.model.element.AnnotationValue` and `AnnotationValueVisitor`
- `javax.lang.model.element.Element` and `ElementVisitor`
- `javax.lang.model.type.TypeMirror` and `TypeVisitor`
- `java.nio.file.FileVisitor` and `SimpleFileVisitor`
- `javax.faces.component.visit.VisitContext` and `VisitCallback`

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 Visitor when you need to perform an operation on all elements of a complex object structure (for example, an object tree).

2. Use the pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others.

Pros and Cons

- ✓ A visitor object can accumulate some useful information while working with various objects. This might be handy when you want to traverse some complex object structure, such as an object tree, and apply the visitor to each object of this structure.

- ✘ You need to update all visitors each time a class gets added to or removed from the element hierarchy.
- ✘ Visitors might lack the necessary access to the private fields and methods of the elements that they're supposed to work with.

Summary

In this lesson, you should have learned how to:

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