Behavioral Design Patterns

Nov 19,2023

Software Engineering

Dr.Abbaszadeh

Yeganeh Azmoudeh , Kimia Omrani ,
Amirhosein Nikzad , Alireza Samanipur

Index

Part 1

Introduction , Chain of Responsibility and Command

Yeganeh Azmoudeh - 97463102

Introduction

What is a design pattern?

Behavioral Design Patterns

VS

Algorithm

Defines a clear set of action.

Design Pattern

It gives a general concept of solving

A particular problem.

(implementation is up to you)

Design Patterns

Design patterns are categorized.

Creational

provide object creation mechanisms that increase flexibility and reuse of existing code.

Structural

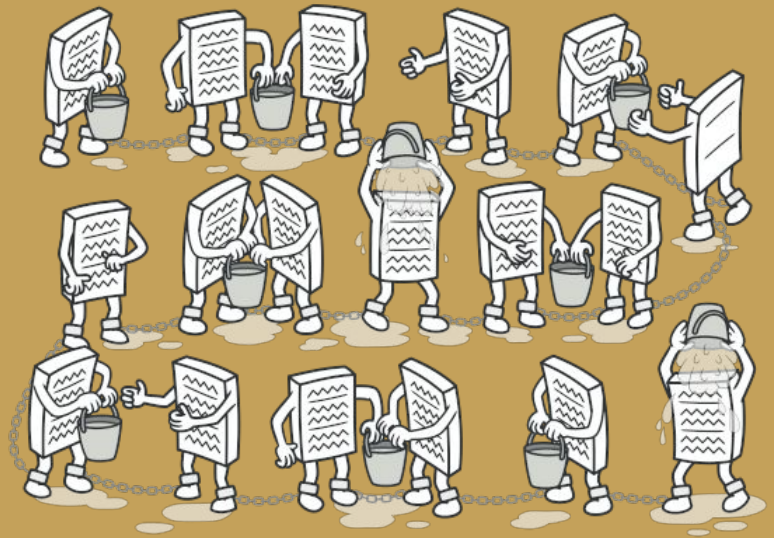
explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

Behavioral

take care of effective communication and the assignment of responsibilities between objects.

Chain of Responsibility

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.

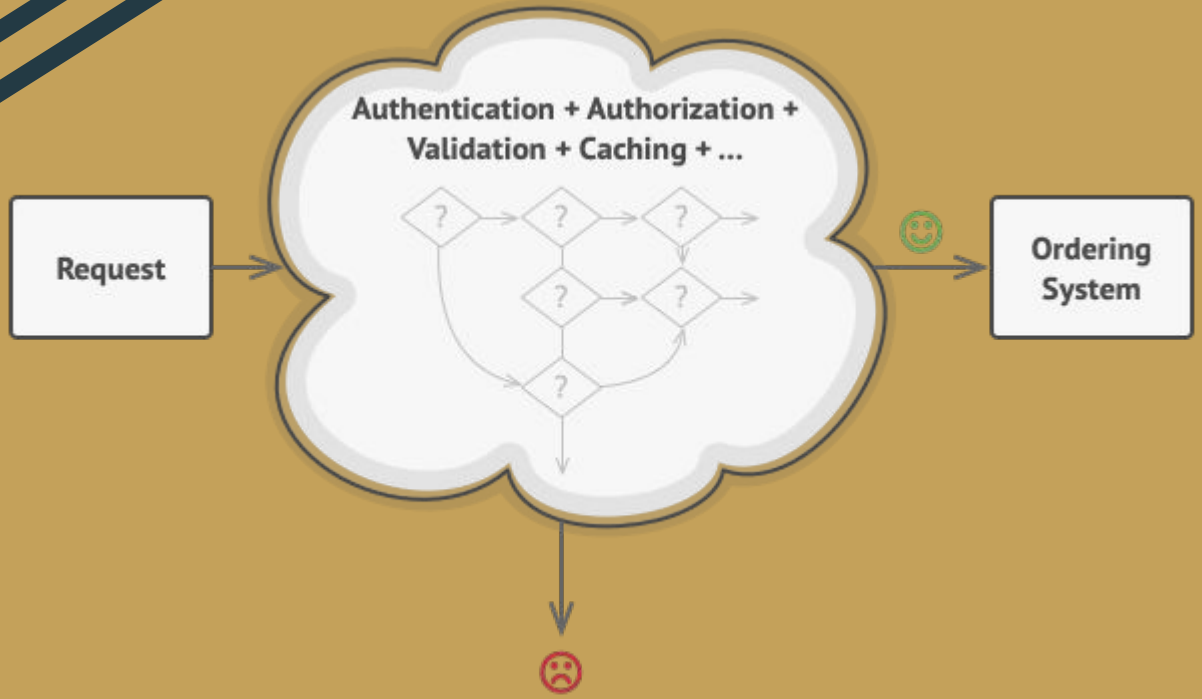


Usage Rate:

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.

Solution ⟶ A series of checks.

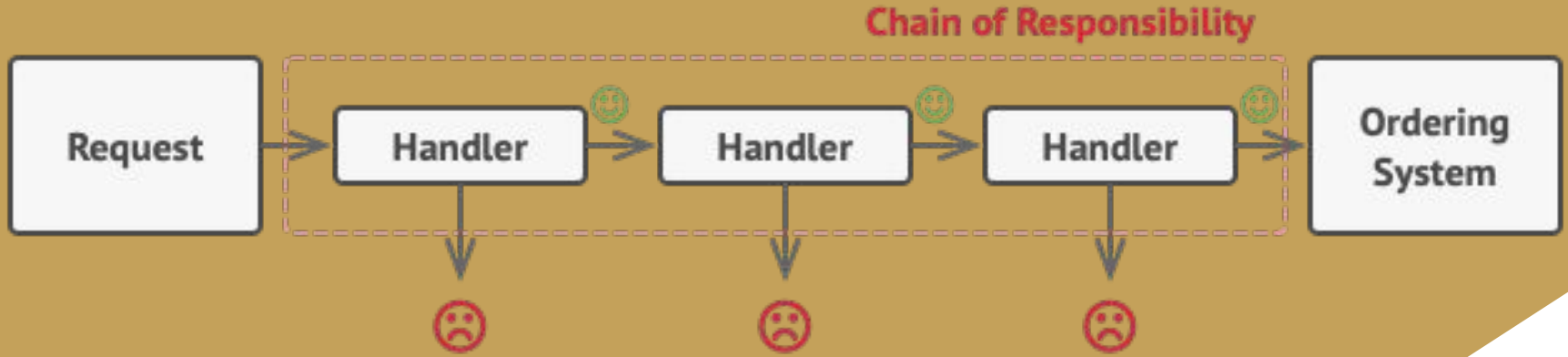


The bigger the code becomes, the messier it becomes.

Solution

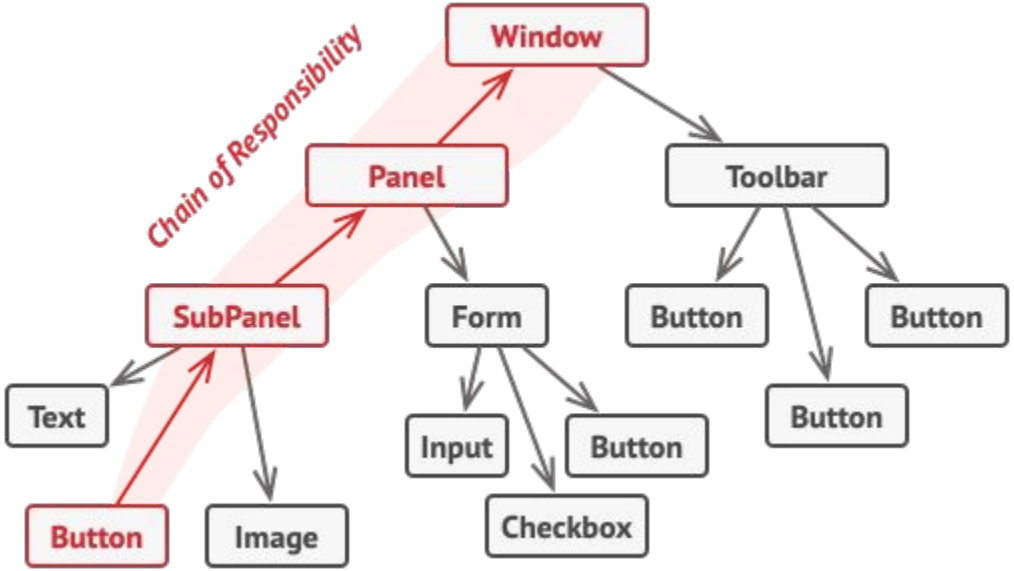
transforming particular behaviors into stand-alone objects called *handlers*. In our case, each check should be extracted to its own class with a single method that performs the check. The request, along with its data, is passed to this method as an argument.

The request travels along the chain until all handlers have had a chance to process it.



Handlers are lined up one by one, forming a chain.

(A handler can decide not to pass the request further down the chain and effectively stop any further processing.)



(GUI elements Tree)

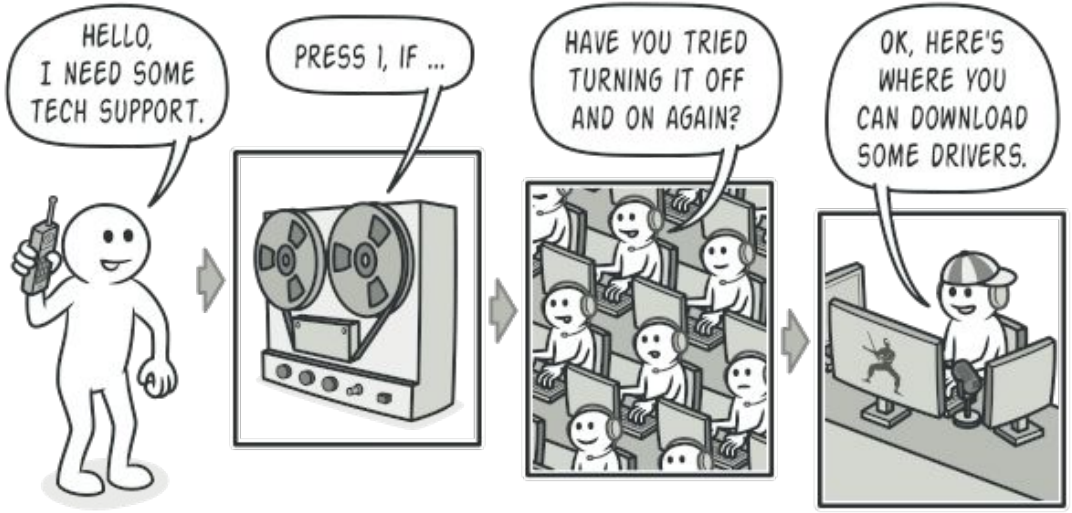
A chain can be formed from a branch of an object tree.

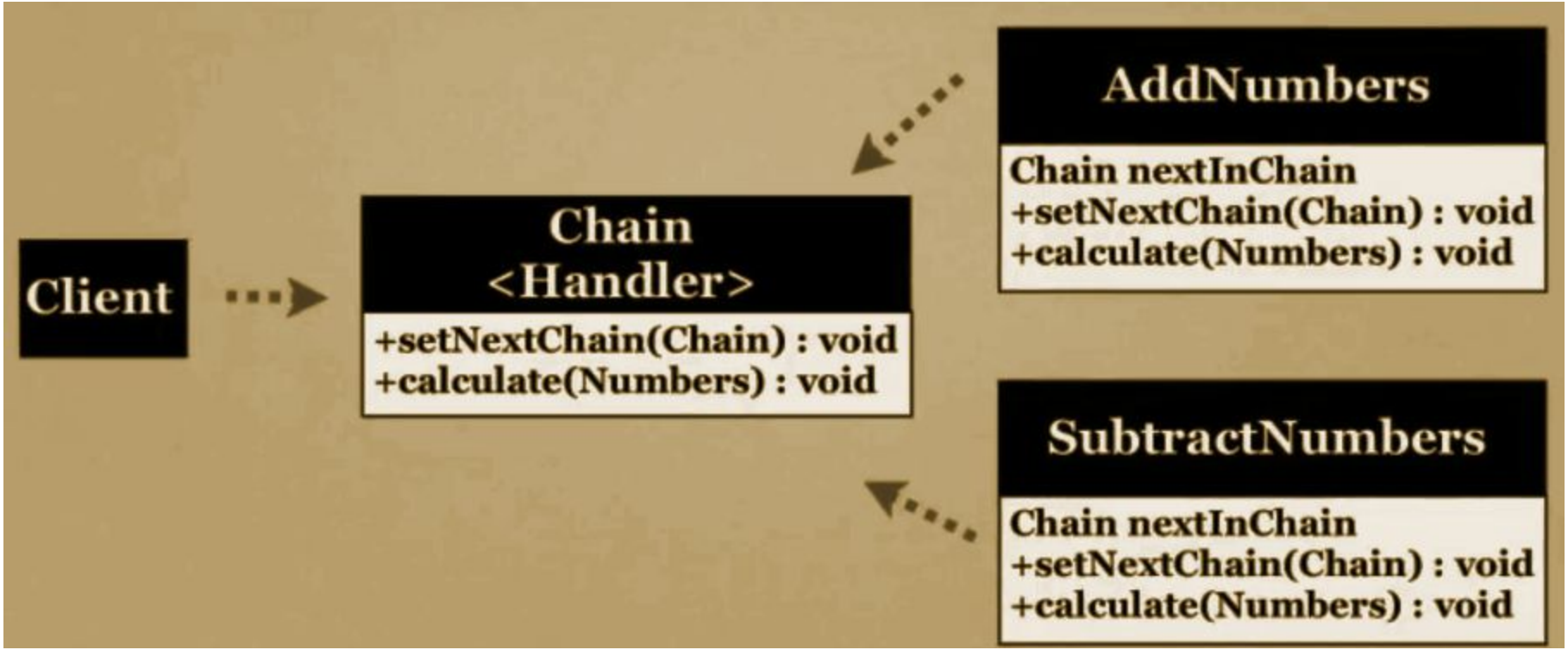
Real-World Analogy

The robotic voice of the autoresponder

→ A live operator

→ A proper engineer




AddNumbers
Chain nextInChain
+setNextChain(Chain) : void
+calculate(Numbers) : void
Client
Chain
<Handler>
+setNextChain(Chain) : void
+calculate(Numbers) : void
SubtractNumbers
Chain nextInChain
+setNextChain(Chain) : void
+calculate(Numbers) : void

(UML Diagram)

Implementation

Calculation example:

Inputs: 2 numbers and an command (request)
Output: 1 number as the result of the calculation

Applicability

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.

The pattern lets you link several handlers into one chain and, upon receiving a request, "ask" each handler whether it can process it. This way all handlers get a chance to process the request.

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

Since you can link the handlers in the chain in any order, all requests will get through the chain exactly as you planned.

Use the CoR pattern when the set of handlers and their order are supposed to change at runtime.

If you provide setters for a reference field inside the handler classes, you'll be able to insert, remove or reorder handlers dynamically.

Pros & Cons

You can control the order.

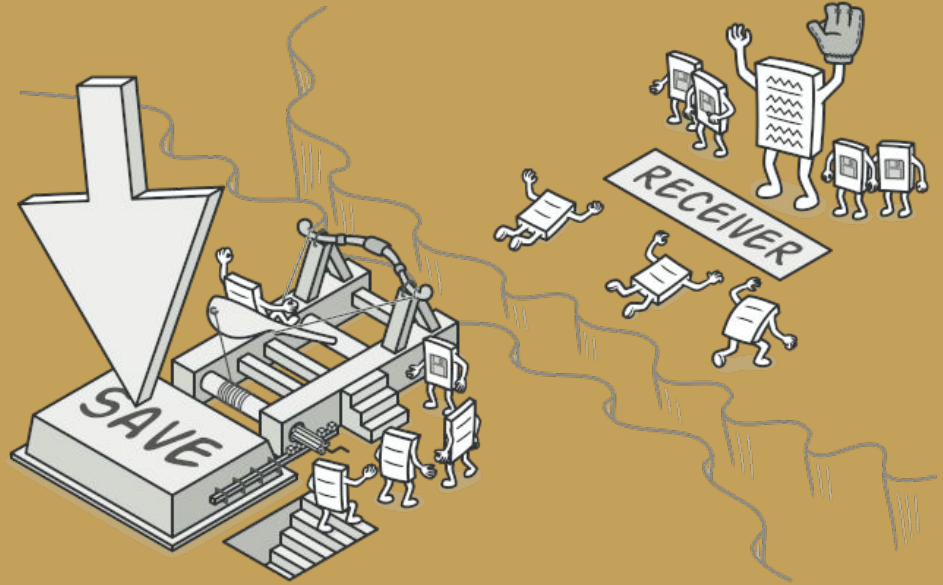
Single Responsibility principle.

Open/Close principle.

Unhandled requests?!

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.

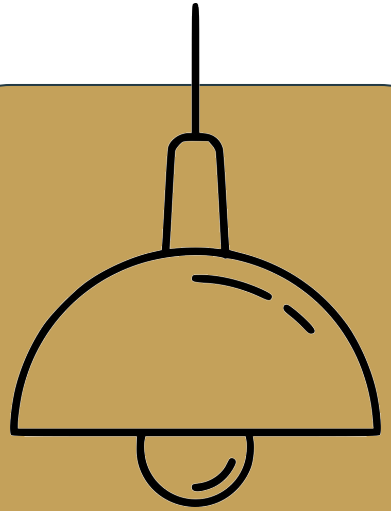


Usage rate:

Sender → Invoker → Receiver

Sender
(A man)

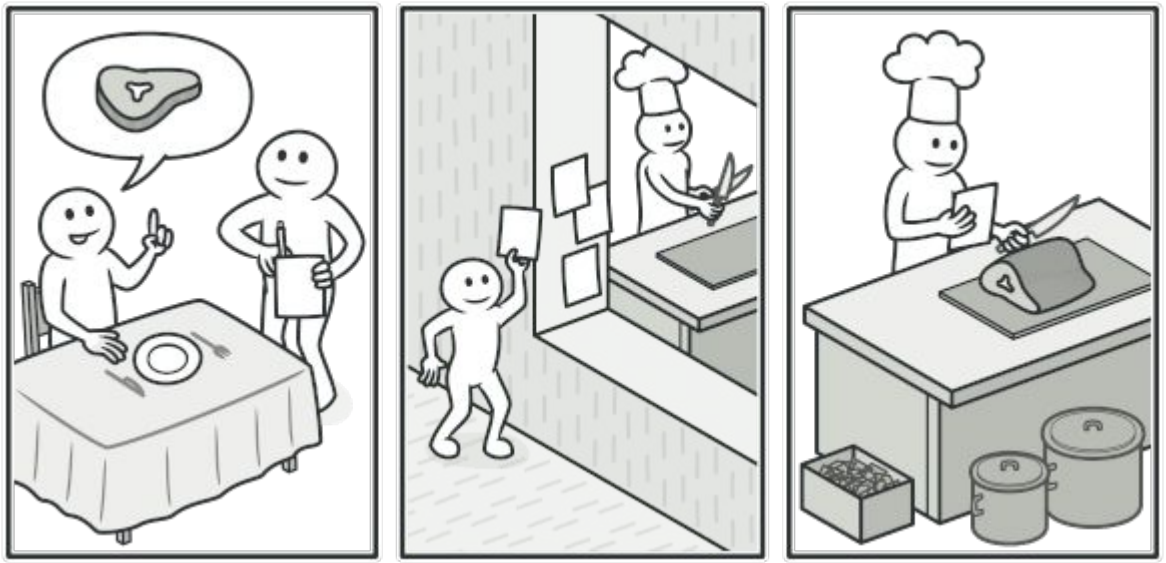
Invoker
(The remote control)

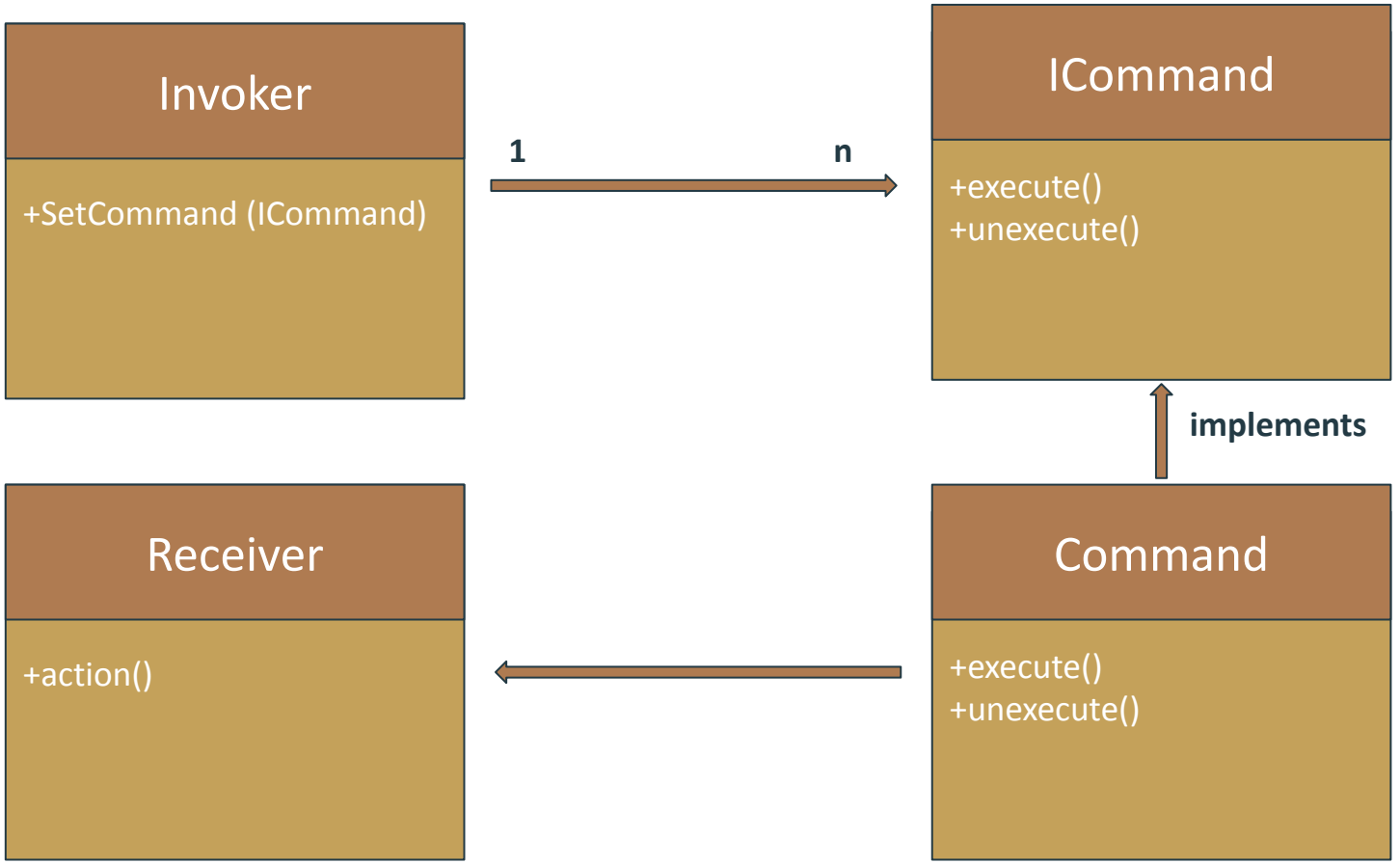


Receiver
(Lamp(s))

Real-World Analogy

Your order is taken (by the waiter)
→ Chef
→ Waiter checks the order and serves it




Invoker
+SetCommand (ICommand)
1
n
ICommand
+execute()
+unexecute()
implements
Receiver
+action()
Command
+execute()
+unexecute()

Implementation

Light Control example:

Input: the main command
Output: action(s)

Pros & Cons

Single Responsibility principle.

Open/Close principle.

Undo and redo.

Assemble a set of commands into one.

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

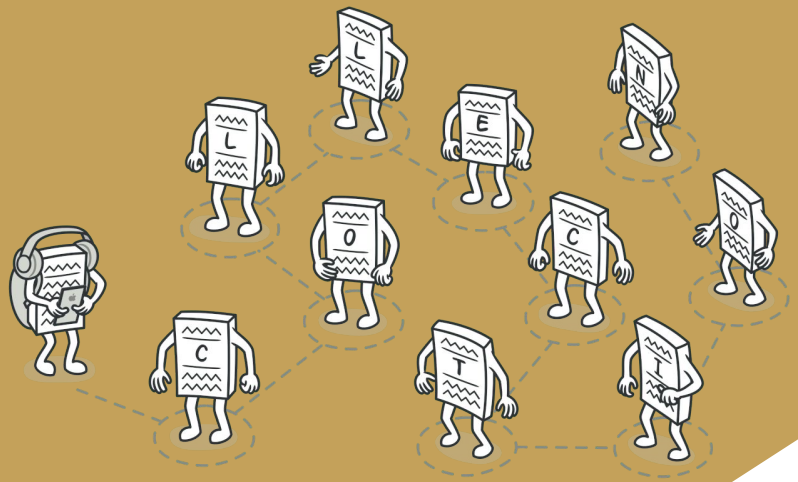
Part 3

Iterator,Mediator and Memento

Kimia Omrani - 97463142

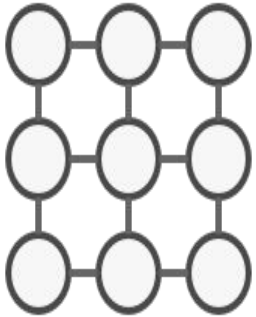
Iterator

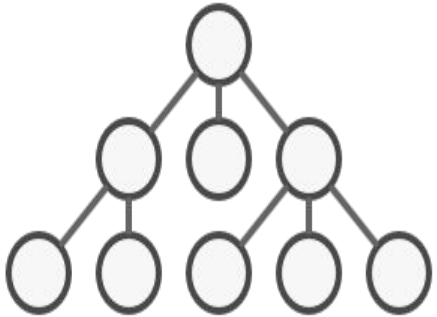
Iterator is a behavioral design pattern that lets you **traverse elements** of a collection **without** exposing its underlying representation(without needing to know the internal structure of the collection likelist, stack, tree, etc.).



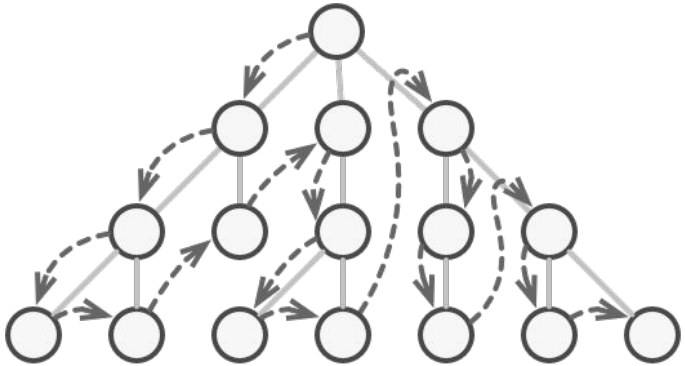
Problem

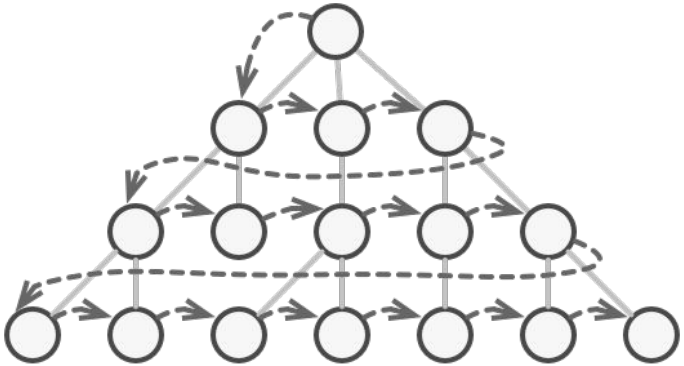
But no matter **how** a collection is **structured**, it must provide some **way of accessing** its elements so that other code can use these elements. There should be a way to **go through each element** of the collection without accessing the same elements over and over.





This may sound like an **easy** job if you have a collection based on a **list**. You just loop over all of the elements. But how do you sequentially traverse elements of a **complex data** structure, such as a tree? For example, one day you might be just fine with depth-first traversal of a tree. Yet the next day you might require breadth-first traversal. And the next week, you might need something else, like random access to the tree elements.





-Adding more and more traversal algorithms to the collection gradually **blurs** its primary responsibility, which is efficient data storage.

-Additionally, some algorithms might be tailored for a **specific** application, so including them into a generic **collection class** would be weird.

-On the other hand, the client code that's supposed to work with various collections may not even **care** how they store their elements.

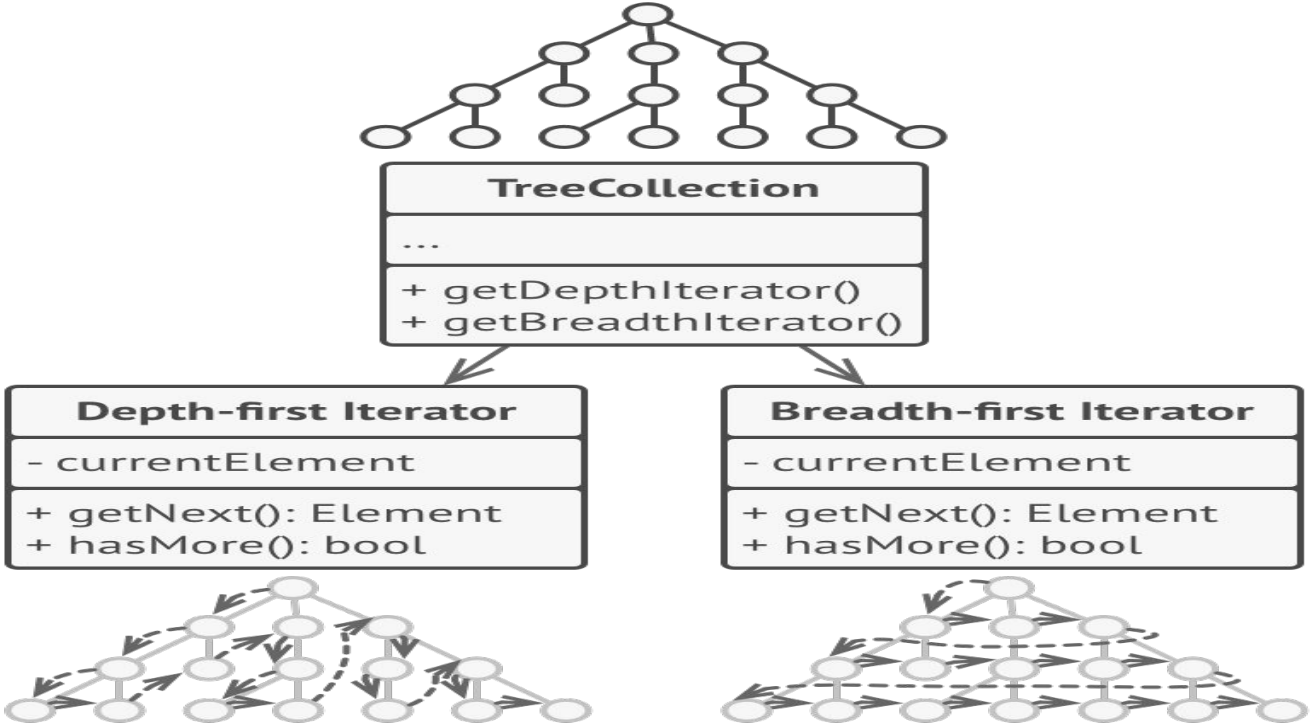
Solution

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

1.In addition to implementing the algorithm itself, an iterator object **encapsulates** all of the traversal details, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection at the same time, independently of each other.

3.All iterators must implement the **same interface.** This makes the client code **compatible** with any collection type or any traversal algorithm as long as there's a proper iterator. If you need a special way to traverse a collection, you just create a new iterator class, **without** having to change the collection or the client.

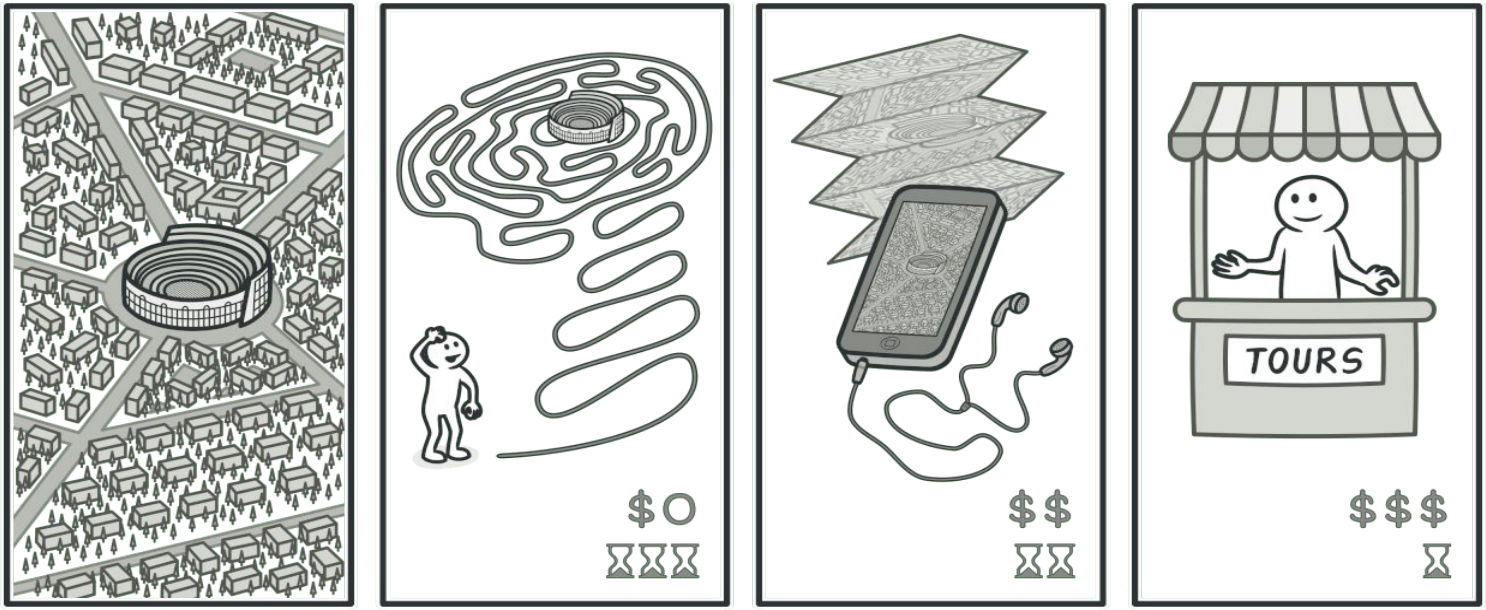
2.iterators provide **one primary method** for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which **means** that the iterator has **traversed** all of the elements.



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

Real-World Analogy

All of these options—the random directions born in your head, the smartphone navigator or the human guide—act as iterators over the vast collection of sights and attractions located in



Applicability

-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).

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

-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

Pros

+ You can **clean up the client code** and the **collections** by extracting bulky traversal algorithms into separate classes.

+ 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.

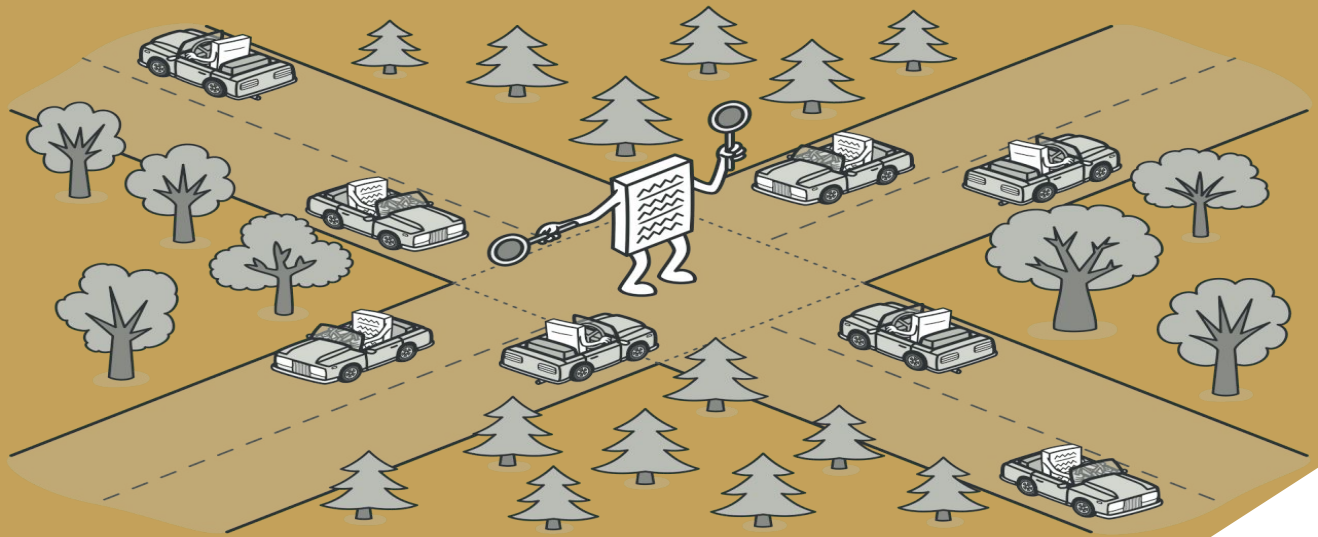
+For the same reason, you can **delay** an iteration and continue it when needed.

Cons

-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



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

without a mediator:

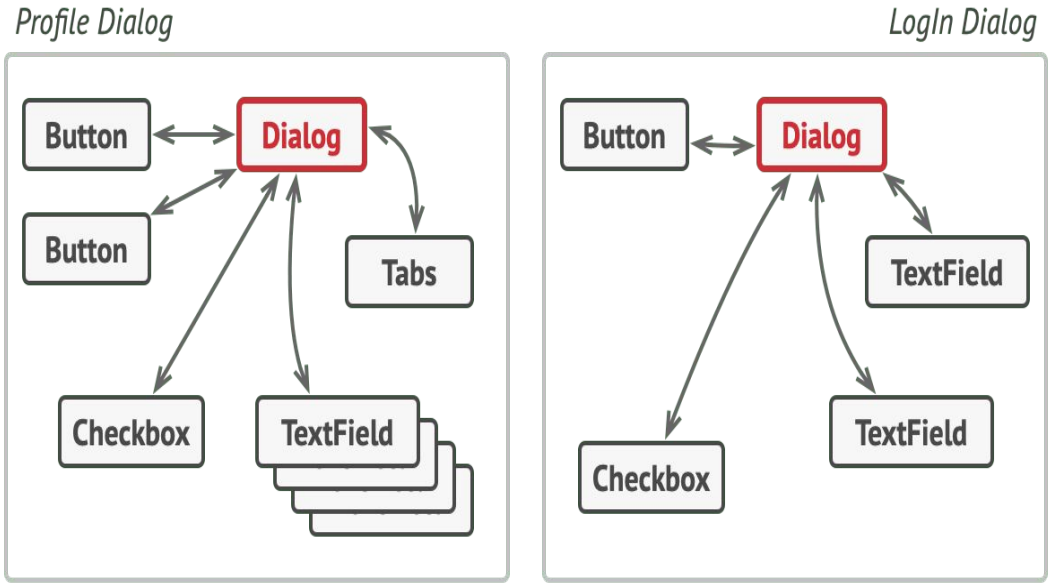
Objects need to have **direct knowledge of each other** in order to communicate. where **changes in one Object** may require modifications in **multiple other objects**. This can make the system more complex, Less maintainable, and harder to extend.

This decentralized approach can make it **difficult to manage** and **control** the interactions between objects.
-Each object may have its own way of communicating with others

Difficulty in adding new objects: When a new object needs to be added to the system, all existing objects would need to be **aware** of and **communicate** directly with the new object.

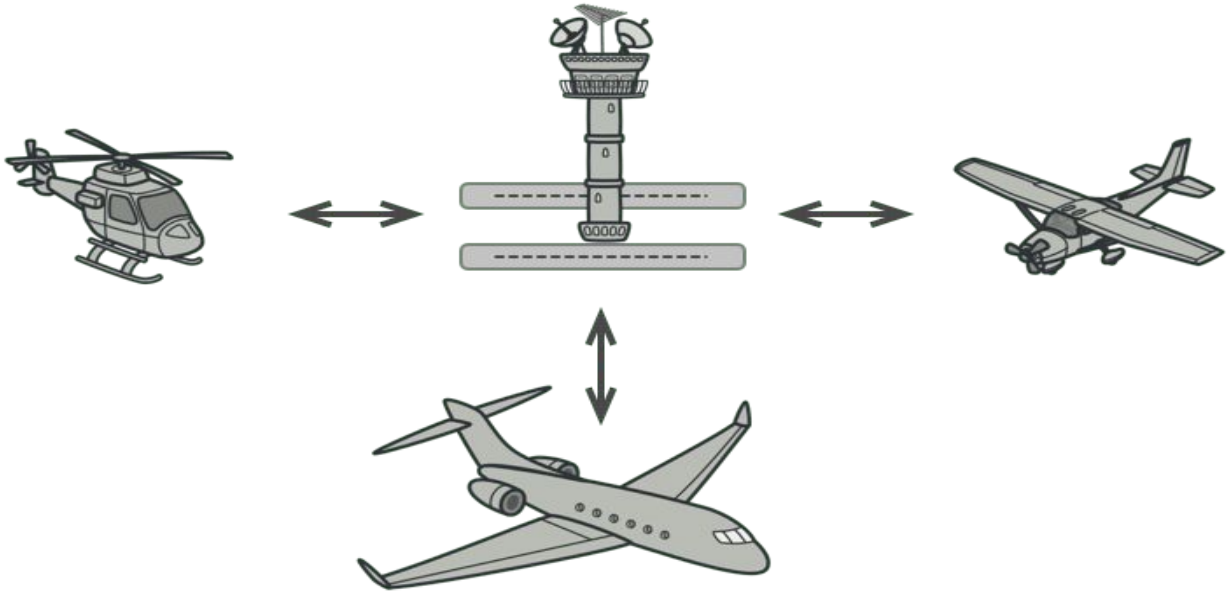
Solution

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.
As a result, the components depend only on a single mediator class i**nstead of** being coupled to dozens of their colleagues.



This way, the Mediator pattern lets you **encapsulate** a complex web of relations between various objects inside a single mediator object.
The **fewer dependencies** a class has, the easier it becomes to modify, extend or reuse that 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.

Applicability

-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.

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

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

Pros and Cons

Pros

You can **extract the communications** between various components into a single place, making it easier to comprehend and maintain.

You can **introduce new mediators** without having to change the actual components.

You can **reduce coupling**.

You can **reuse** individual components more easily.

Cons

Over time a mediator can evolve into a God Object.

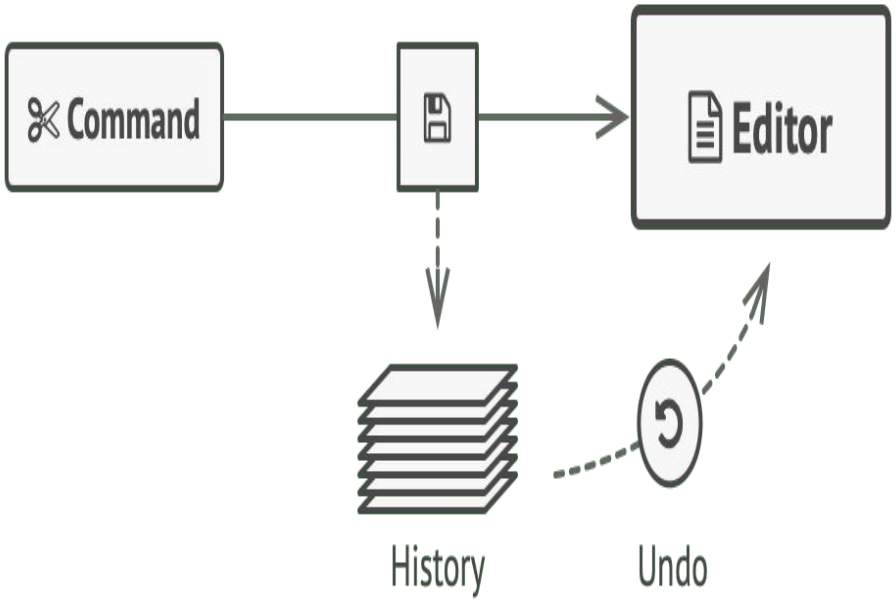
In software development, the term "god object" refers to an **anti-pattern** where a single object or class in a system takes on an excessive amount of responsibilities or functionality. It violates the principles of good object-oriented design, such as the Single Responsibility Principle (SRP), which states that a class should have only one reason to change.

Memento

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

Also known as: Snapshot

Problem



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.

Actual “snapshots” of the editor’s state

What data does it contain?

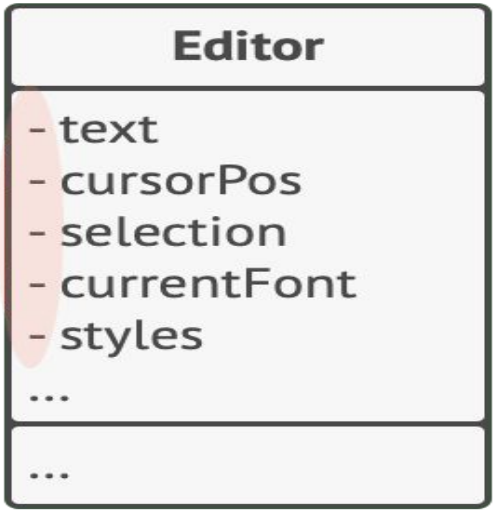
actual text,
cursor coordinates,
current scroll position, etc

To make a snapshot, you’d need to **collect** these values and put them into some kind of **container**.

It looks like we’ve reached a dead end: you either expose all internal details of classes, making them too fragile, or restrict access to their state, making it impossible to produce snapshots. Is there any other way to implement the "undo"?

private = can’t copy

public = unsafe



Solution

All problems that we've just experienced are caused by broken encapsulation.

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.

Storing the **copy of the object's state** in a special object called ***memento***. The contents of the memento aren't accessible to any other object **except** the one that produced it. Other objects **must communicate with mementos** using a limited interface which may allow fetching the snapshot's metadata (creation time, the name of the performed operation, etc.), but not the original object's state contained in the snapshot.

Such a restrictive policy lets you **store** mementos inside other objects, usually called ***caretakers***. Since the caretaker works with the memento only via the limited interface, it's not able to tamper with the state stored inside the memento. At the same time, the **originator** has access to all fields inside the memento, allowing it to restore its previous state at will.

Applicability

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.

While most people remember this pattern thanks to the "undo" use case, it's also indispensable when dealing with transactions (i.e., if you need to roll back an operation on error).

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

The Memento makes the object itself responsible for creating a snapshot of its state. No other object can read the snapshot, making the original object's state data safe and secure.

Pros and Cons

Pros

+You can produce snapshots of the object's state without violating its encapsulation.
+You can **simplify** the originator's code by letting the **caretake**r maintain the **history of the originator**'s state.

Cons

-The app might consume lots of RAM if
-Caretakers should track the originator's **lifecycle** to be able to destroy obsolete mementos.
-Most dynamic programming languages, such as PHP, Python and JavaScript, can't guarantee that the state within the memento stays **untouched**.

Part 3

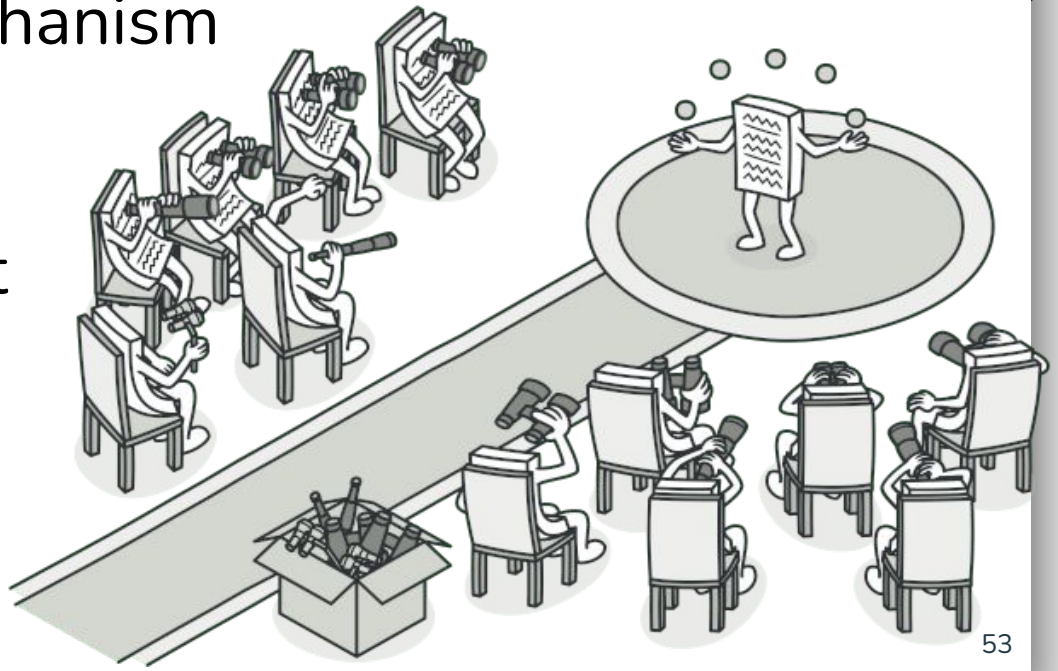
Observer , State and Strategy

Amirhosein Nikzad - 99463182

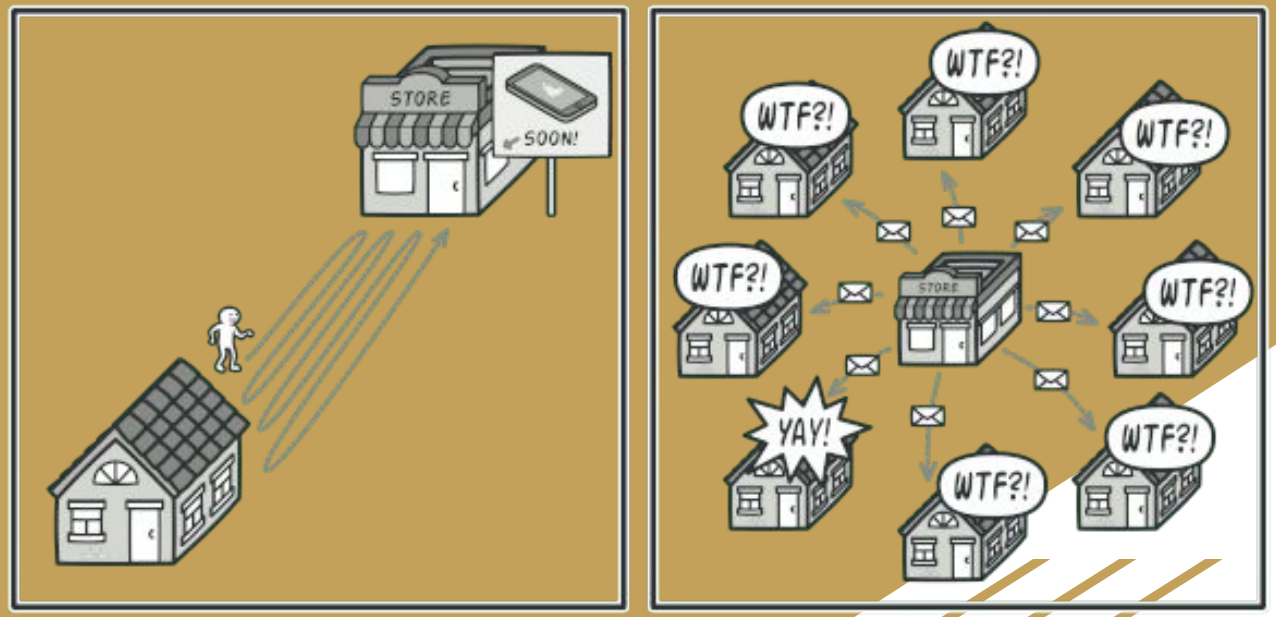
Observer

Observer

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.

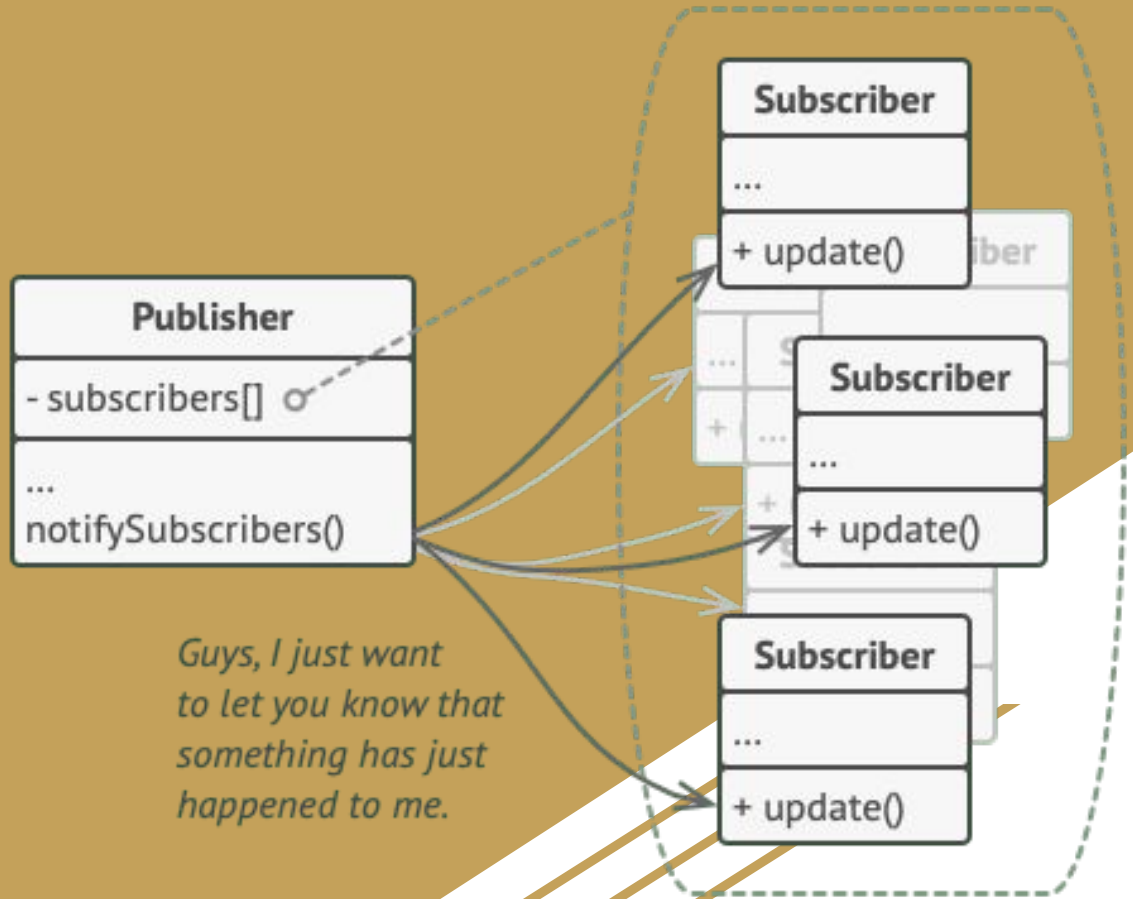


Problem



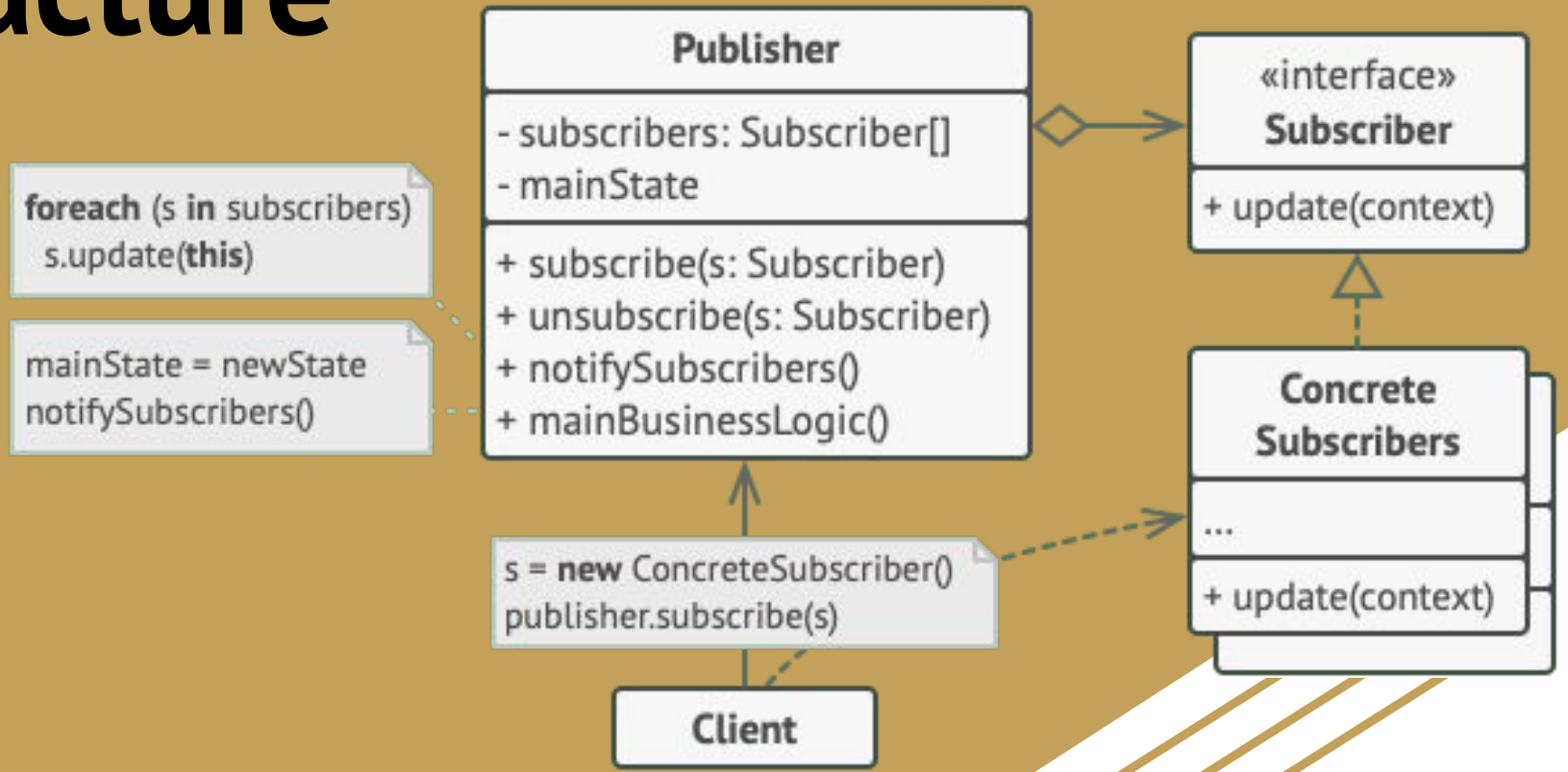
Solution


Hey, sign me up, please!
Subscriber
Subscriber
Me too!
Publisher
- subscribers[]
...
+ addSubscriber(subscriber)
+ removeSubscriber(subscriber)


Publisher
- subscribers[]
...
notifySubscribers()
Guys, I just want to let you know that something has just happened to me.
Subscriber
...
+ update()
Subscriber
...
+ update()
Subscriber
...
+ update()

Real-World Analogy

Structure


Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
«interface»
Subscriber
+ update(context)
foreach (s in subscribers)
s.update(this)
mainState = newState
notifySubscribers()
Concrete
Subscribers
...
+ update(context)
s = new ConcreteSubscriber()
publisher.subscribe(s)
Client

Applicability

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.

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

The subscription list is dynamic, so subscribers can **join or leave** the list whenever they need to.

Pros and Cons

Open/Closed Principle.

You can introduce new subscriber classes without having to change the publisher's code

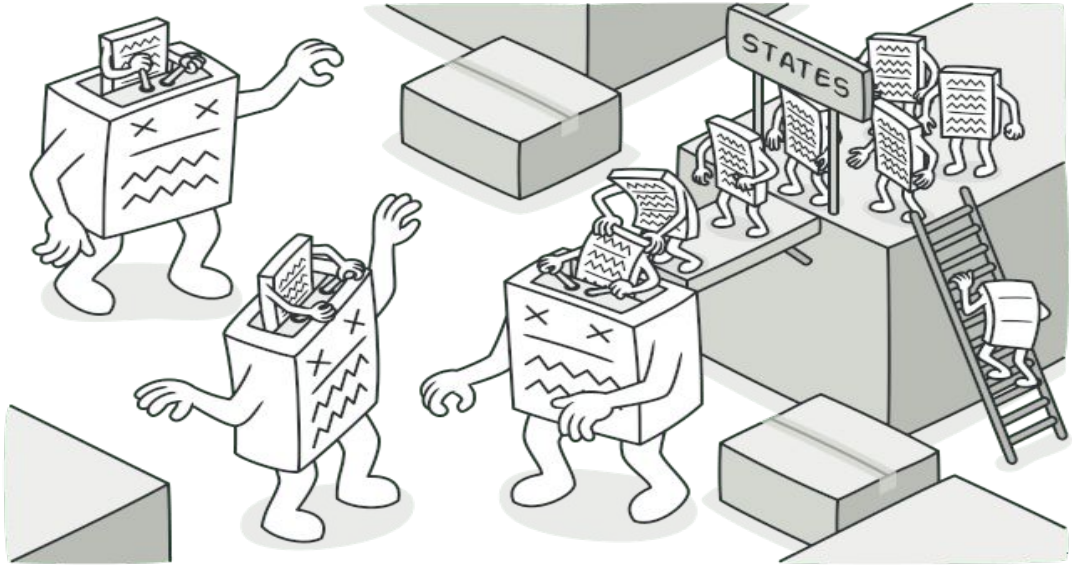
You can **establish relations** between objects **at runtime**.

Subscribers are notified in **random order**

State

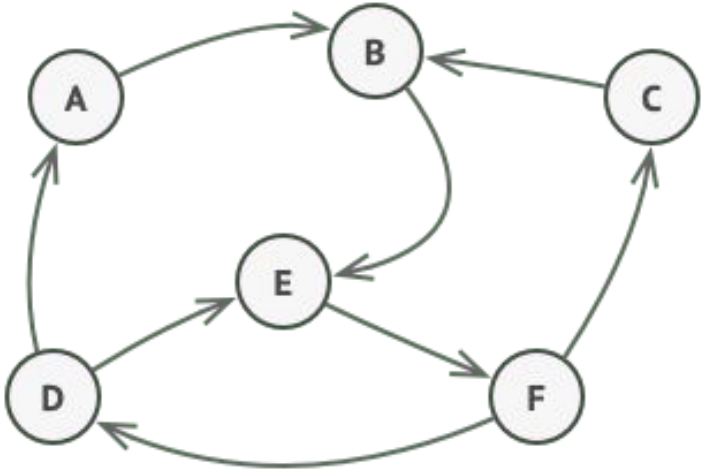
State

State is a behavioral design pattern that lets an **object alter its behavior when its internal state changes**.
It appears as
if the object
changed its class.



Problem

The main idea is that, at **any given moment, there's a finite number of states which a program can be in**.
with in any **unique state, program behaves differently**, and program can be switched from one state to another instantaneously.



Problem

Draft

it moves the document to moderation.

Moderation

it makes the document public but only if the current user is an administrator.

Published

it doesn't do anything at all.

finite-state

```
class Document is
    field state: string
    // ...
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == "admin")
                    state = "published"
                break
            "published":
                // Do nothing.
                break
    // ...
```

Problem

State machines are usually implemented with **lots of conditional statements** (if or switch) that select the **appropriate behavior depending on the current state** of the object.

The biggest weakness of a state machine based on **conditionals reveals** itself once we start **adding more and more states and state-dependent behaviors** to the Document class.

Solution

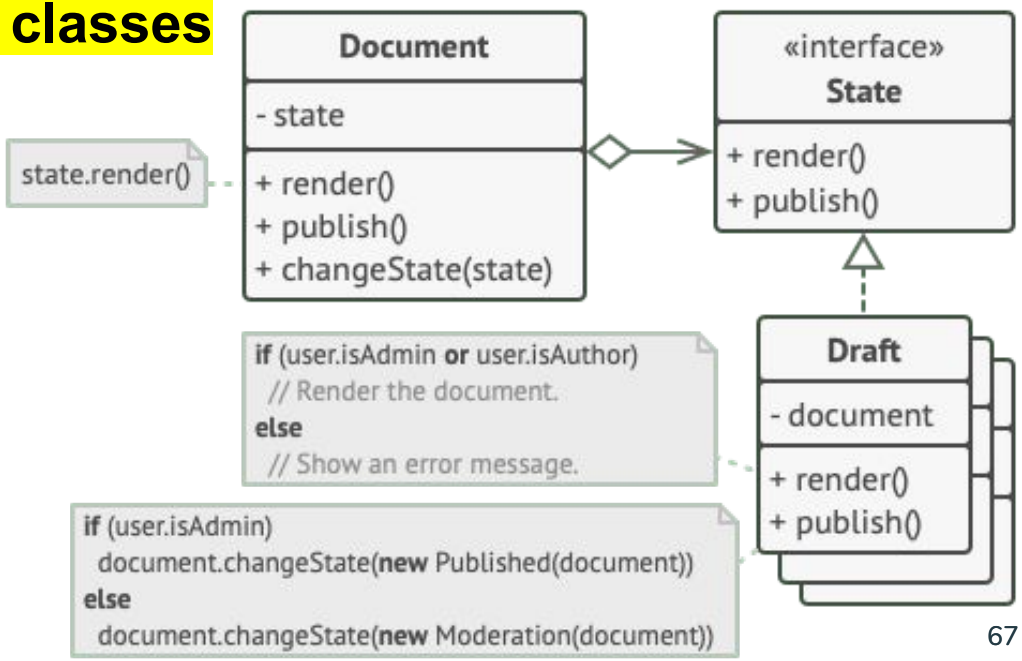
The State pattern suggests that you **create new classes for all possible states** of an object and **extract all state-specific behaviors into these classes**.

Instead of implementing all behaviors on its own, the original object, called **context**, **stores a reference to one of the state objects that represents its current state**, and **delegates all the state-related work to that object**.

Solution

To transition the context into another state, **replace the active state object with another object that represents that new state**.

This is possible only if all s**tate classes follow the same interface** and the context itself works with these objects through that interface.

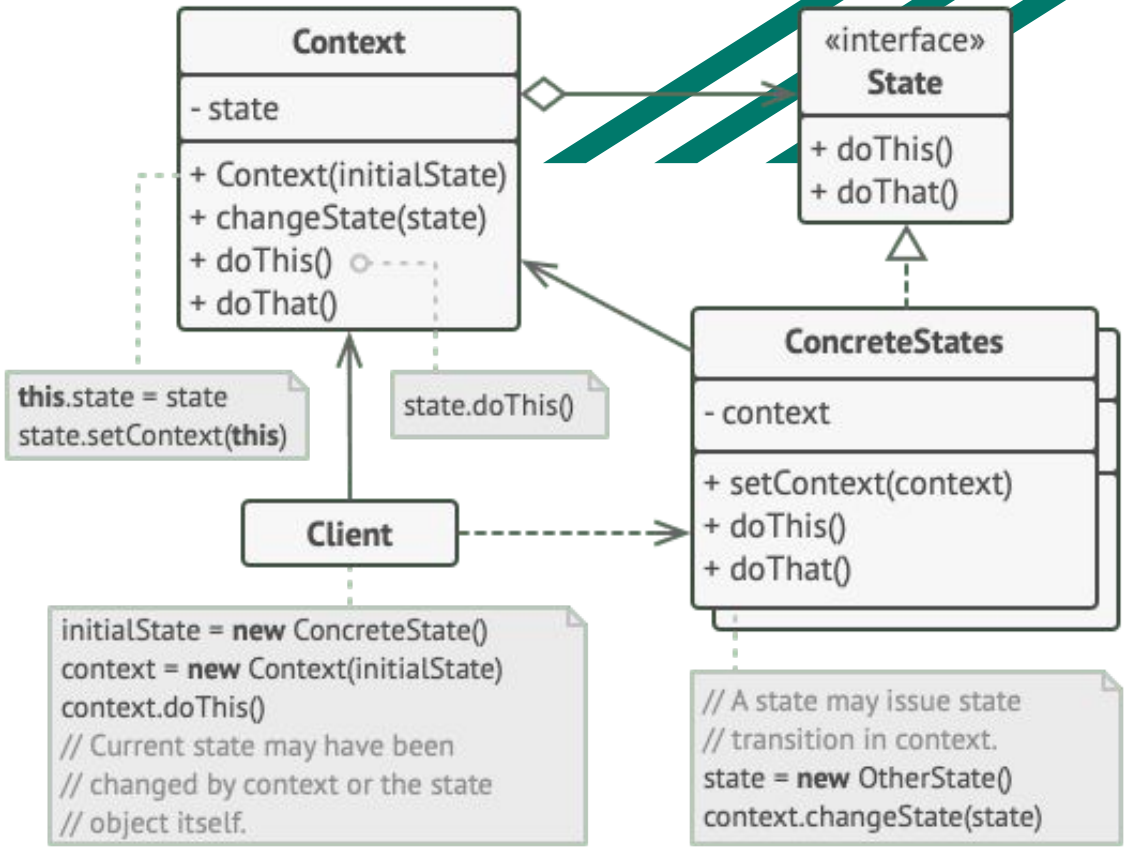


State vs Strategy

This structure may look similar to the Strategy pattern, but there's one key difference.

In the State pattern, the particular states may be aware of each other and initiate transitions from one state to another,whereas strategies almost never know about each other.

Structure


Context
- state
+ Context(initialState)
+ changeState(state)
+ doThis()
+ doThat()
«interface»
State
+ doThis()
+ doThat()
ConcreteStates
- context
+ setContext(context)
+ doThis()
+ doThat()
Client
this.state = state
state.setContext(this)
state.doThis()
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// Current state may have been
// changed by context or the state
// object itself.
// A state may issue state
// transition in context.
state = new OtherState()
context.changeState(state)

Applicability

Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is big, and the state-specific code changes frequently.

The pattern suggests that you extract all state-specific code into a set of distinct classes. As a result, you can add new states or change existing ones independently of each other, reducing the maintenance cost.

Applicability

Use the pattern when you have a class polluted with massive conditionals that alter how the class behaves according to the current values of the class's fields.

Use State when you have a lot of duplicate code across similar states and transitions of a condition-based state machine.

Pros and Cons

Single Responsibility Principle

Organize the code related to particular states into separate classes

Open/Closed Principle

Introduce new states without changing existing state classes or the context

Simplify the code of the context by eliminating bulky state machine conditionals

Applying the pattern can be overkill if a state machine has only a few states or rarely changes

Strategy

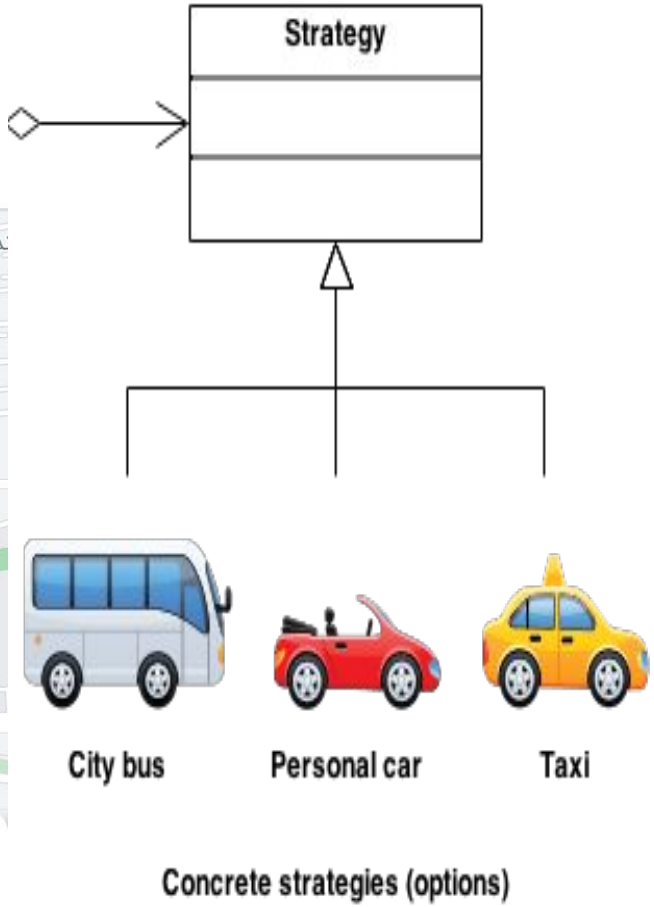
Strategy

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

Navigation app
route planning
Walking routes
Public transport
Cyclists
Car
Taxi

Navigator maintenance issues



Problem

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.

Teamwork became inefficient

Your teammates, who had been hired right after the successful release, complain that they spend too much time resolving merge conflicts. Implementing a new feature requires you to change the same huge class, conflicting with the code produced by other people.

Solution

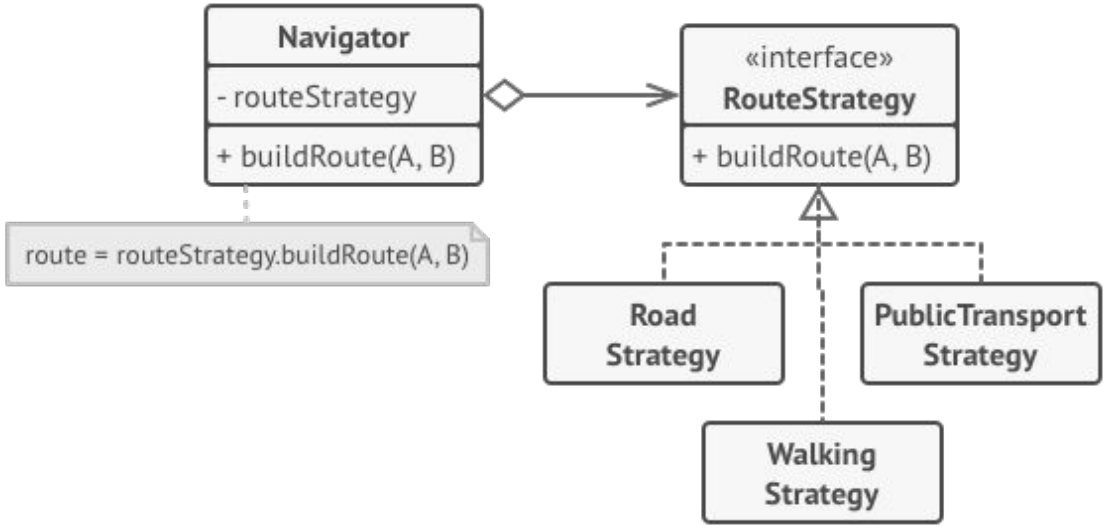
Take a class that does something specific in a lot of different ways and **extract all of these algorithms into separate classes called strategies**.

The original class(called context)**delegates the work to a linked strategy object** instead of executing it on its own.

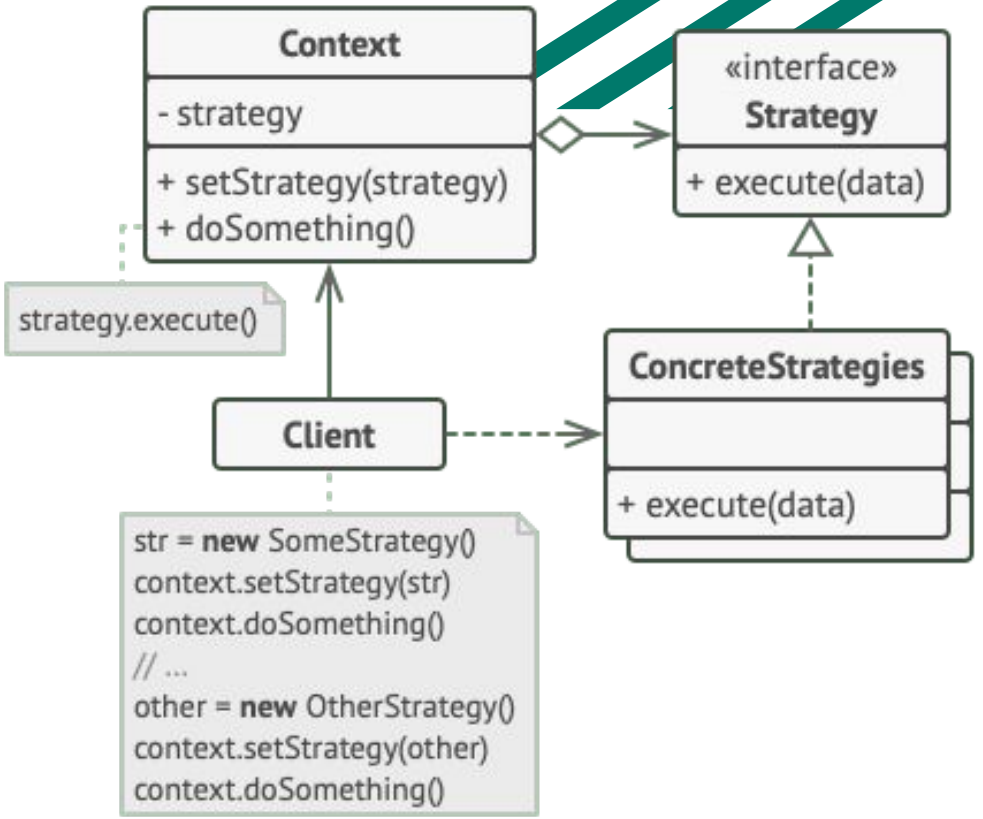
The client passes the desired strategy to the context.

Solution

This way the context becomes independent of concrete strategies, so you can add new algorithms or modify existing ones without changing the code of the context or other strategies.



Structure


Context
- strategy
+ setStrategy(strategy)
+ doSomething()
«interface»
Strategy
+ execute(data)
strategy.execute()
Client
ConcreteStrategies
+ execute(data)
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()

Applicability

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.

When you have a **lot of similar classes that only differ in the way they execute some behavior**.

Applicability

The Strategy pattern lets you extract the varying behavior into a separate class hierarchy and combine the original classes into one, thereby reducing duplicate code.

The Strategy pattern lets you isolate the code, internal data, and dependencies of various algorithms from the rest of the code. Clients get a simple interface to execute the algorithms and switch them at runtime.

Pros

You can swap algorithms used inside an object at runtime.

Open/Closed Principle

You can introduce new strategies without having to change the context

You can isolate the implementation details of an algorithm from the code that uses it.

You can replace inheritance with composition

Cons

If you **only have a couple of algorithms** and they rarely change

Clients must be **aware of the differences between strategies** to be able to select a proper one

A lot of **modern programming** languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. You can use it without bloating your code with extra classes and interfaces

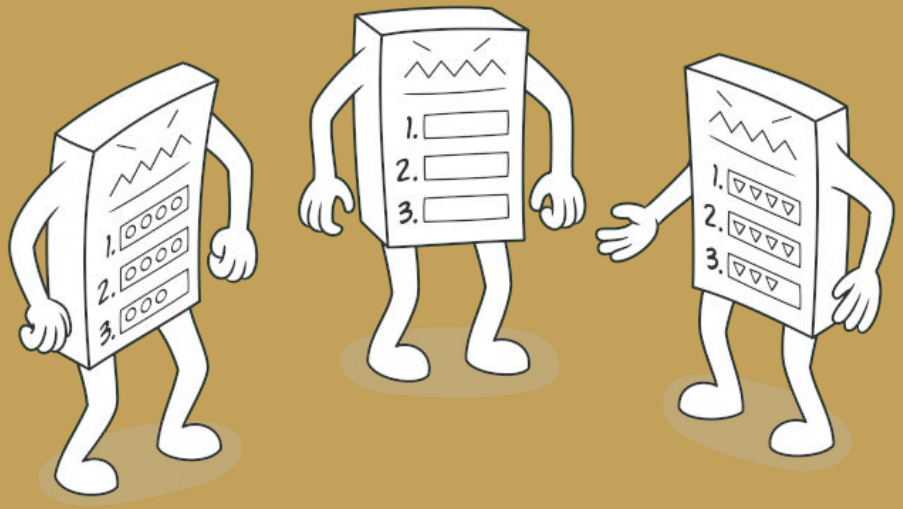
Part 4

Template method & visitor

Alireza Samanipour - 98463129

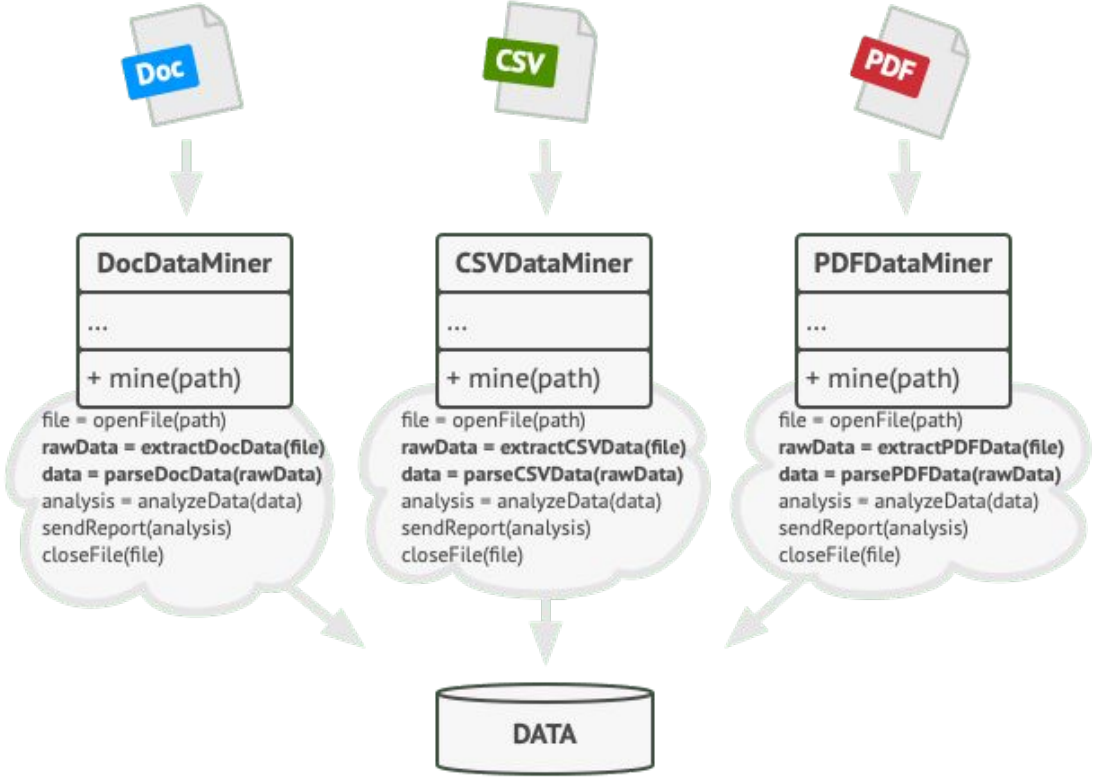
Template method

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.



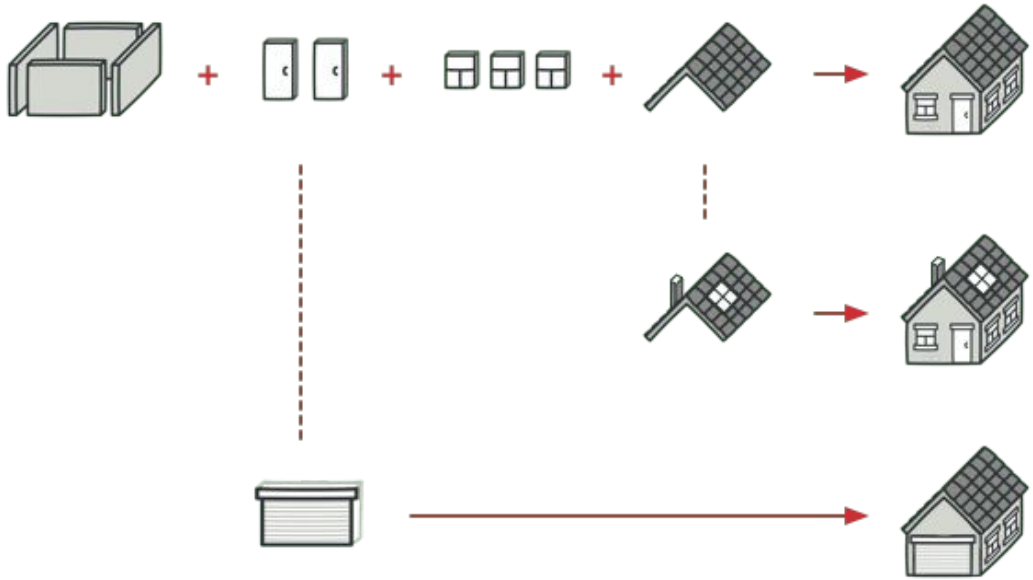
Usage rate:

Problem


Doc
CSV
PDF
DocDataMiner
...
+ mine(path)
file = openFile(path)
rawData = extractDocData(file)
data = parseDocData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
CSVDataMiner
...
+ mine(path)
file = openFile(path)
rawData = extractCSVData(file)
data = parseCSVData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
PDFDataMiner
...
+ mine(path)
file = openFile(path)
rawData = extractPDFData(file)
data = parsePDFData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
DATA

Real-World Analogy

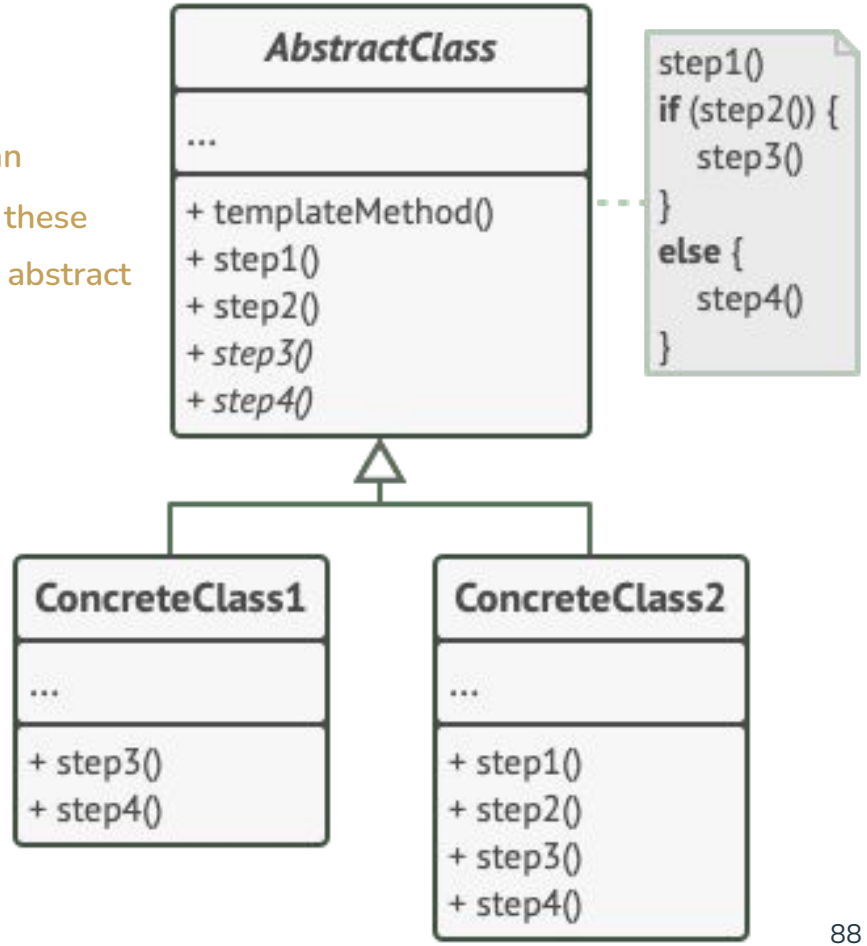
Construction of the foundation
Building walls
Construction of internal walls
Making windows
Finishing the house



Structure

The **Abstract Class** declares methods that act as steps of an algorithm, as well as the actual template method which calls these methods in a specific order. The steps may either be declared abstract or have some default implementation.

Concrete Classes can override all of the steps, but not the template method itself.



Implementation

```
1   abstract class HouseTemplate {
2           public void BuildHouse(){
3               BuildFoundation();
4               BuildWalls();
5               BuildInnerWalls();
6               BuildWindows();
7               BuildCompleted();
8           }
9           protected abstract void BuildFoundation();
10          protected abstract void BuildWalls();
11          protected abstract void BuildInnerWalls();
12          protected abstract void BuildWindows();
13          protected abstract void BuildCompleted();
14      }
```

Implementation

```
1  class Woodenhouse : HouseTemplate{
2      protected override void BuildCompleted(){
3          Console.WriteLine("Wooden House Built");
4      }
5
6      protected override void BuildFoundation(){
7          Console.WriteLine("Wooden House Foundation Built");
8      }
9
10     protected override void BuildInnerWalls(){
11         Console.WriteLine("Wooden House Inner Walls Built");
12     }
13
14     protected override void BuildWalls(){
15         Console.WriteLine("Wooden House Walls Built");
16     }
17
18     protected override void BuildWindows(){
19         Console.WriteLine("Wooden House Windowses Built");
20     }
21 }
```

Implementation

```
class BricksHouse : HouseTemplate{
    protected override void BuildCompleted(){
        Console.WriteLine("Bricks House Built");
    }

    protected override void BuildFoundation(){
        Console.WriteLine("Bricks House Foundation Built");
    }

    protected override void BuildInnerWalls(){
        Console.WriteLine("Bricks House Inner Walls Built");
    }

    protected override void BuildWalls(){
        Console.WriteLine("Bricks House Walls Built");
    }

    protected override void BuildWindows(){
        Console.WriteLine("Bricks House Windowses Built");
    }
}
```

Implementation

```
static void Main(string[] args){
    try{
            BricksHouse bricksHouse = new BricksHouse();
            bricksHouse.BuildHouse();

            Woodenhouse woodenhouse = new Woodenhouse();
            woodenhouse.BuildHouse();
    }catch (Exception ex){
                ShowError(ex.Message);
    }
    Console.ReadLine();
}
```

Applicability

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.

The Template Method lets you turn a monolithic algorithm into a series of individual steps which can be easily extended by subclasses while keeping intact the structure defined in a superclass.

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.

When you turn such an algorithm into a template method, you can also pull up the steps with similar implementations into a superclass, eliminating code duplication. Code that varies between subclasses can remain in subclasses.

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.

You might violate the *Liskov Substitution Principle* by suppressing a default step implementation via a subclass.

Template methods tend to be harder to maintain the more steps they have.

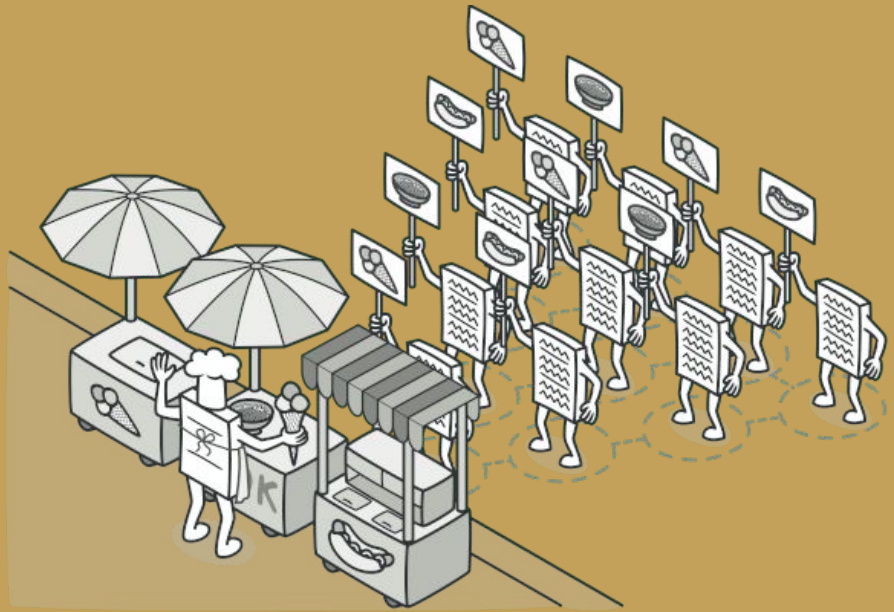
Relations with Other Patterns

- **Factory Method** is a specialization of Template Method. At the same time, a *Factory Method* may serve as a step in a large *Template Method*.

- Template Method is based on inheritance: it lets you alter parts of an algorithm by extending those parts in subclasses. **Strategy** is based on composition: you can alter parts of the object's behavior by supplying it with different strategies that correspond to that behavior. *Template Method* works at the class level, so it's static. *Strategy* works on the object level, letting you switch behaviors at runtime.

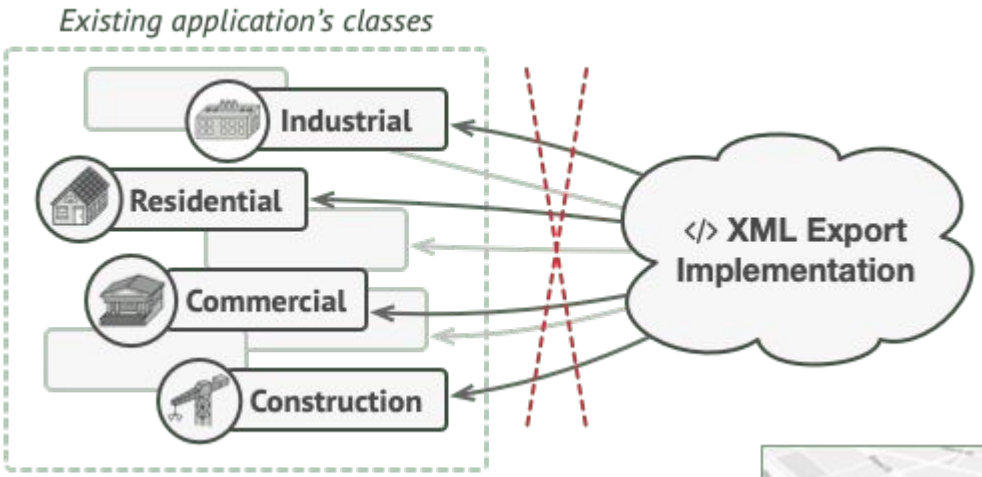
Visitor

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

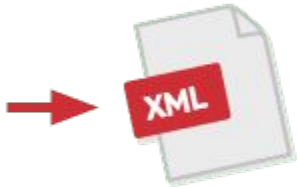


Usage rate:

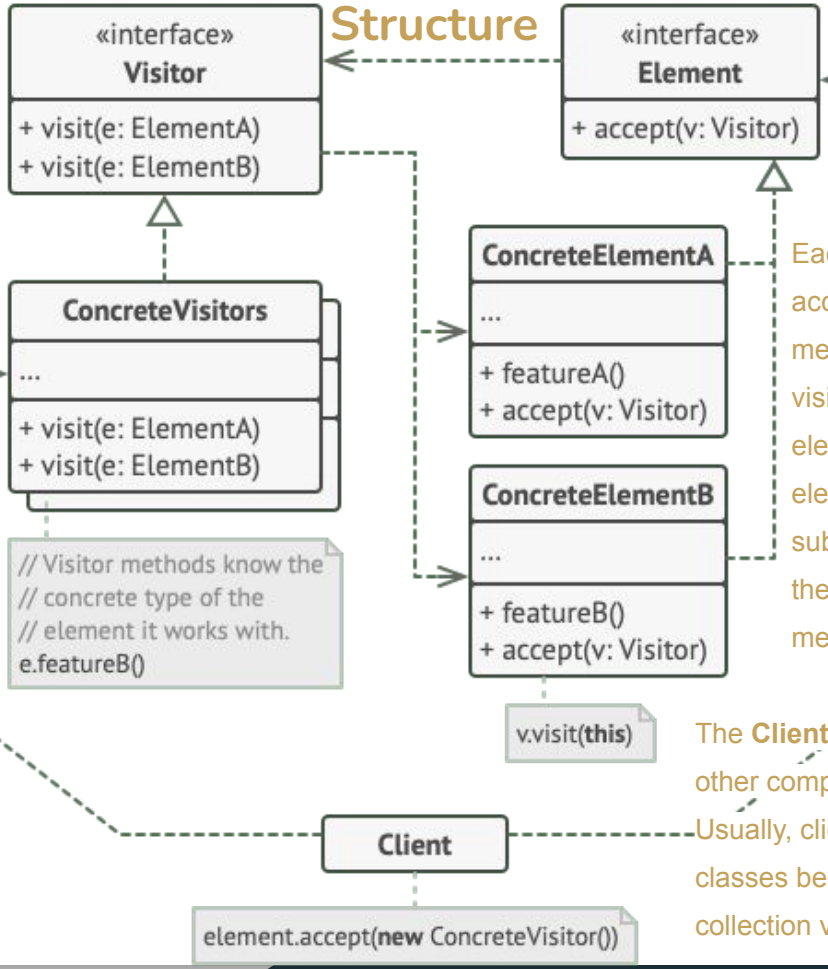
Problem


Existing application's classes
Industrial
Residential
Commercial
Construction
</> XML Export Implementation


LOWER EAST SIDE
CHINATOWN
LITTLE ITALY


XML

Structure



The **Visitor** interface declares a set of visiting methods that can take concrete elements of an object structure as arguments. These methods may have the same names if the program is written in a language that supports overloading, but the type of their parameters must be different.

Each **Concrete Visitor** implements several versions of the same behaviors, tailored for different concrete element classes.

The **Element** interface declares a method for "accepting" visitors. This method should have one parameter declared with the type of the visitor interface.

Each **Concrete Element** must implement the acceptance method. The purpose of this method is to redirect the call to the proper visitor's method corresponding to the current element class. Be aware that even if a base element class implements this method, all subclasses must still override this method in their own classes and call the appropriate method on the visitor object.

The **Client** usually represents a collection or some other complex object (for example, a **Composite** tree). Usually, clients aren't aware of all the concrete element classes because they work with objects from that collection via some abstract interface.

Real-World Analogy

```
1   interface ItemElement{
2       int accept(ShoppingCartVisitor visitor);
3   }
```

Implementation

```
1   class Fruit : ItemElement{
2       private int pricePerKg;
3       private int weight;
4       private String name;
5       public Fruit(int priceKg, int wt, String nm){
6           this.pricePerKg = priceKg;
7           this.weight = wt;
8           this.name = nm;
9       }
10
11      public int getPricePerKg(){
12          return pricePerKg;
13      }
14
15      public int getWeight(){
16          return weight;
17      }
18
19      public String getName(){
20          return this.name;
21      }
22
23      public int accept(ShoppingCartVisitor visitor){
24          return visitor.visit(this);
25      }
26  }
```

Implementation

```
1   class Book : ItemElement{
2       int price;
3       private String isbnNumber;
4       public Book(int cost, String isbn){
5           this.price = cost;
6           this.isbnNumber = isbn;
7       }
8
9       public int getPrice(){
10          return price;
11      }
12
13      public String getIsbnNumber(){
14          return isbnNumber;
15      }
16
17      public int accept(ShoppingCartVisitor visitor){
18          return visitor.visit(this);
19      }
20  }
```

Implementation

```
1   class ShoppingCartVisitorImpl : ShoppingCartVisitor{
2       public int visit(Book book){
3           int cost = 0;
4           if (book.getPrice() > 50){
5               cost = book.getPrice() - 5;
6           }else
7               cost = book.getPrice();
8               Console.WriteLine("Book ISBN::" + book.getIsbnNumber() + " cost =" + cost);
9               return cost;
10          }
11
12      public int visit(Fruit fruit){
13          int cost = fruit.getPricePerKg() * fruit.getWeight();
14          Console.WriteLine(fruit.getName() + " cost = " + cost);
15          return cost;
16      }
17  }
```

Implementation

```
1   static void Main(string[] args){
2       try{
3           ItemElement[] items = new ItemElement[] { new Book(20, "1234"),
4               new Book(100, "5678"), new Fruit(10, 2, "Banana"),
5               new Fruit(5, 5, "Apple") };
6           int total = calculatePrice(items);
7
8           Console.WriteLine("Total Cost = " + total);
9       }catch (Exception ex){
10          ShowError(ex.Message);
11      }
12      Console.ReadLine();
13  }
14  private static int calculatePrice(ItemElement[] items){
15      ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
16      int sum = 0;
17      foreach (ItemElement item in items){
18          sum = sum + item.accept(visitor);
19      }
20      return sum;
21  }
22
```

Applicability

Use the Visitor when you need to perform an operation on all elements of a complex object structure (for example, an object tree).
The Visitor pattern lets you execute an operation over a set of objects with different classes by having a visitor object implement several variants of the same operation, which correspond to all target classes.

Use the Visitor to clean up the business logic of auxiliary behaviors.
The pattern lets you make the primary classes of your app more focused on their main jobs by extracting all other behaviors into a set of visitor classes.

Use the pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others.
You can extract this behavior into a separate visitor class and implement only those visiting methods that accept objects of relevant classes, leaving the rest empty.

Pros and Cons

Open/Closed Principle. You can introduce a new behavior that can work with objects of different classes without changing these classes.

Single Responsibility Principle. You can move multiple versions of the same behavior into the same class.

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.

Relations with Other Patterns

- You can treat Visitor as a powerful version of the Command pattern. Its objects can execute operations over various objects of different classes.
- You can use Visitor to execute an operation over an entire Composite tree.
- You can use Visitor along with Iterator to traverse a complex data structure and execute some operation over its elements, even if they all have different classes.

References

1. https://refactoring.guru/design-patterns/behavioral-patterns
2. https://www.youtube.com/@derekbanas
3. https://www.youtube.com/@christopherokhravi
4. https://holosen.net/