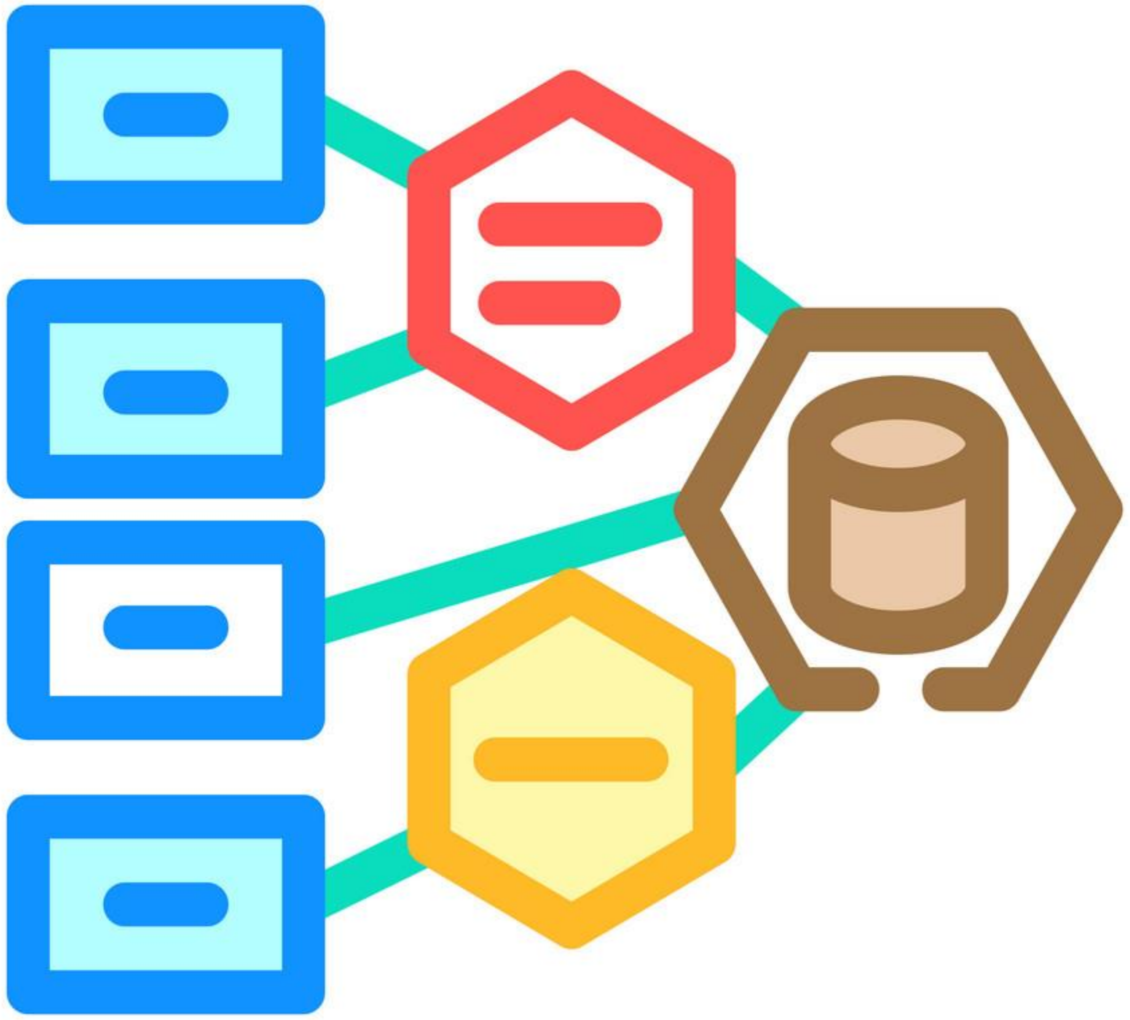
Software Design and architecture

Fall 2024

Dr. Natalia Chaudhry



Creational patterns

- Factory
- Abstract factory
- Builder
- Prototype
- Singleton

Structural patterns

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral patterns

- Chain of responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

A good link to refer: https://refactoring.guru/design-patterns

Behavioral patterns

- ✓ Chain of responsibility
- ✓ Command
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template method
- ✓ Visitor

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.

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.

- 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.

The system became very hard to comprehend and expensive to maintain. You struggled with the code for a while, until one day you decided to refactor the whole thing.

- **Chain of Responsibility** relies on transforming particular behaviors into stand-alone objects called handlers.
- 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.

Steps to Implement:

1. **Define a handler interface** with a method to handle requests and set the next handler.
2. **Create concrete handlers**, each responsible for specific requests.
3. **Chain handlers** together in order.
4. **Pass requests** to the chain's start point.

A logging system has **Error**, **Warning**, and **Info** levels.

Handler Interface:

```
class Logger:
    def __init__(self, level):
        self.level = level
        self.next = None

    def set_next(self, next_logger):
        self.next = next_logger

    def log(self, message, severity):
        if severity >= self.level:
            self.write(message)
        elif self.next:
            self.next.log(message, severity)

    def write(self, message):
        raise NotImplementedError
```

Concrete Handlers

```
class ErrorLogger(Logger):
    def write(self, message):
        print(f"ERROR: {message}")

class WarningLogger(Logger):
    def write(self, message):
        print(f"WARNING: {message}")

class InfoLogger(Logger):
    def write(self, message):
        print(f"INFO: {message}")
```

Chain Setup

```
error_logger = ErrorLogger(3)
warning_logger = WarningLogger(2)
info_logger = InfoLogger(1)

error_logger.set_next(warning_logger)
warning_logger.set_next(info_logger)
```

Usage

```
error_logger.log("This is an error.", 3)  # Handled by ErrorLogger
error_logger.log("This is a warning.", 2)  # Handled by
WarningLogger
error_logger.log("This is an info message.", 1)  # Handled by
InfoLogger
```

- 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 (If a request is passed through the chain and none of the handlers is capable of processing it, the request goes unhandled.).

Behavioral patterns

- ✓ Chain of responsibility
- ✓ Command
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template method
- ✓ Visitor

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request.

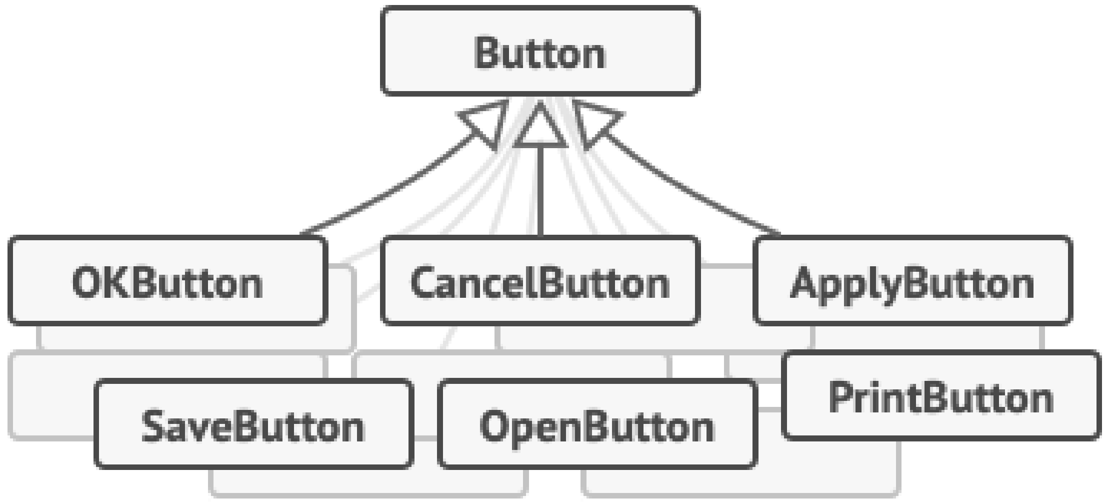
This transformation lets you pass requests as a method arguments, delay or queue a request’s execution, and support undoable operations.

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.

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.

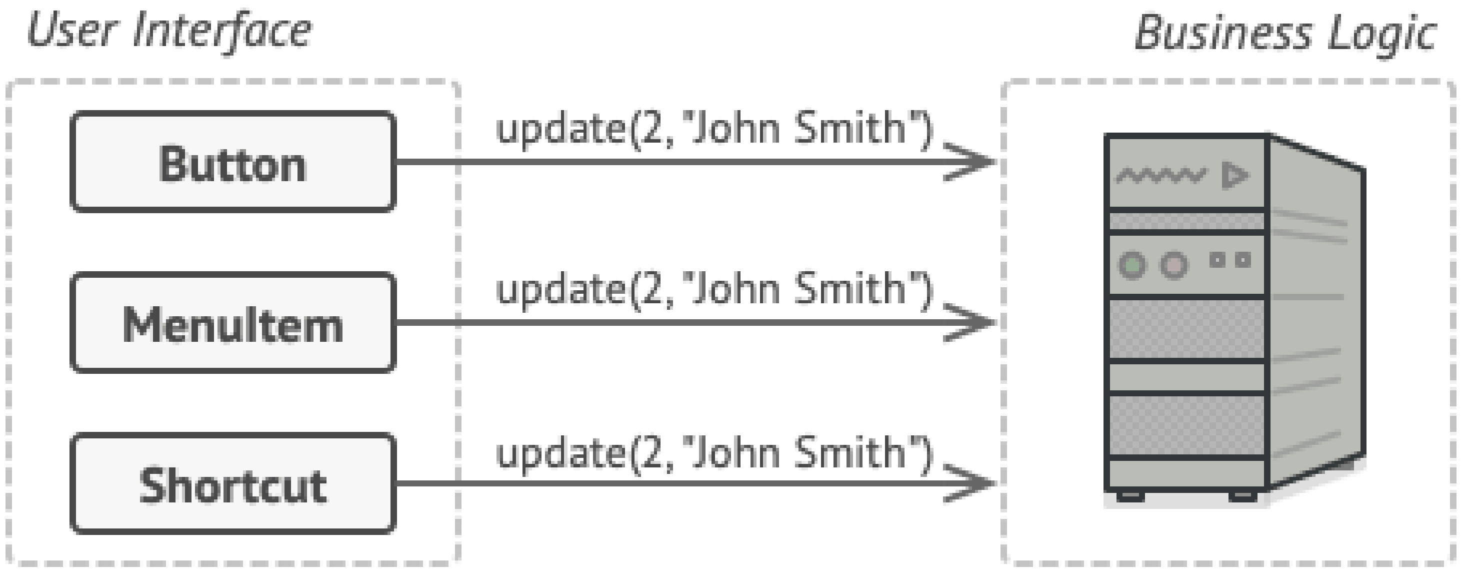


First, you have an enormous number of subclasses, and that would be okay if you weren’t risking breaking the code in these subclasses each time you modify the base Button class.

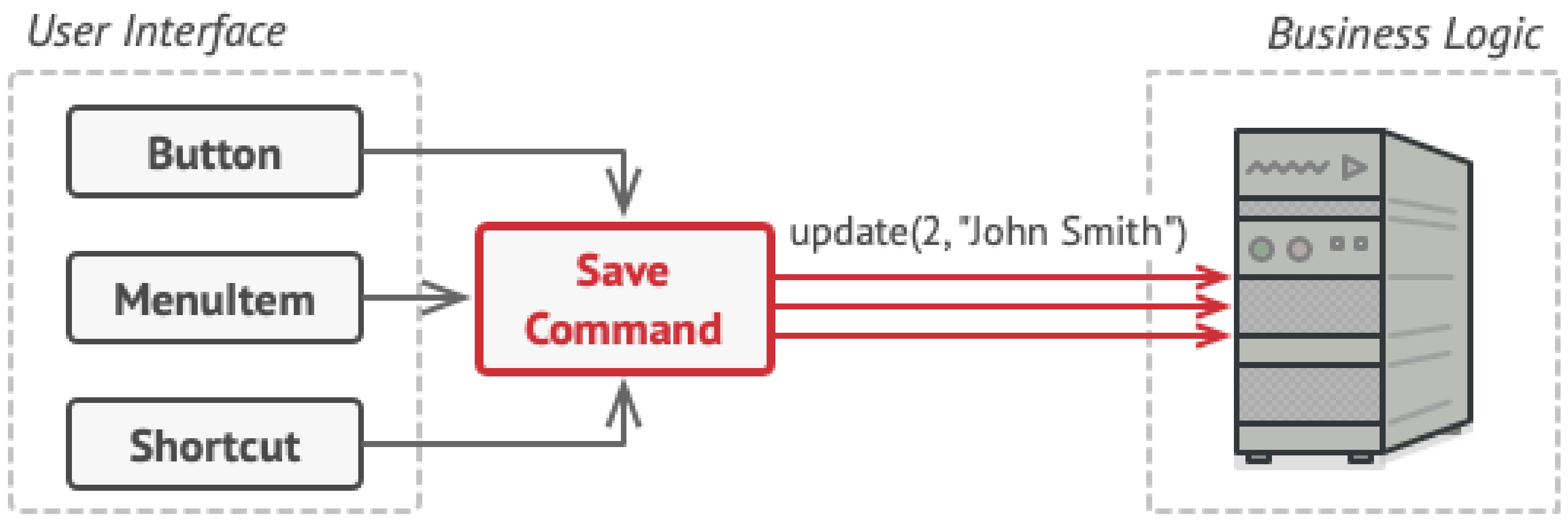
Put simply, your GUI code has become awkwardly dependent on the volatile code of the business logic.

- Good software design is often based on the principle of separation of concerns, which usually results in breaking an app into layers.
- The most common example: a layer for the graphical user interface and another layer for the business logic.
- The GUI layer is responsible for rendering a beautiful picture on the screen, capturing any input and showing results of what the user and the app are doing.
- However, when it comes to doing something important, like calculating the trajectory of the moon or composing an annual report, the GUI layer delegates the work to the underlying layer of business logic.

- In the code it might look like this: 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.


User Interface
Button
MenuItem
Shortcut
Save Command
update(2, "John Smith")
Business Logic

Steps to Implement:

- Create a command interface with an execute() method.
- Create concrete command classes that implement the interface and encapsulate the specific action.
- Create an invoker class to trigger the command.
- Create a receiver class (the object that performs the action).The invoker triggers the command, and the command then invokes the appropriate method on the receiver.

Home Automation System

Consider a scenario where you have a **light** and a **fan** in a smart home system. You can issue commands to turn them on and off.

Command Interface

```
class Command:
    def execute(self):
        pass
```

Concrete Commands:

```
class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_on()

class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_off()
```

```
class FanOnCommand(Command):
    def __init__(self, fan):
        self.fan = fan

    def execute(self):
        self.fan.turn_on()

class FanOffCommand(Command):
    def __init__(self, fan):
        self.fan = fan

    def execute(self):
        self.fan.turn_off()
```

Receiver Classes (the objects that perform actions):

```
class Light:
    def turn_on(self):
        print("Light is ON")

    def turn_off(self):
        print("Light is OFF")

class Fan:
    def turn_on(self):
        print("Fan is ON")

    def turn_off(self):
        print("Fan is OFF")
```

Invoker Class (responsible for executing commands):

```
class RemoteControl:
    def __init__(self):
        self.command = None

    def set_command(self, command):
        self.command = command

    def press_button(self):
        self.command.execute()
```

Client Code:

```
# Create the receiver objects
light = Light()
fan = Fan()

# Create command objects
light_on = LightOnCommand(light)
light_off = LightOffCommand(light)
fan_on = FanOnCommand(fan)
fan_off = FanOffCommand(fan)

# Create the invoker
remote = RemoteControl()

# Turn light ON and OFF
remote.set_command(light_on)
remote.press_button()  # Light is ON
remote.set_command(light_off)
remote.press_button()  # Light is OFF

# Turn fan ON and OFF
remote.set_command(fan_on)
remote.press_button()  # Fan is ON

remote.set_command(fan_off)
remote.press_button()  # Fan is OFF
```

- 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.

Behavioral patterns

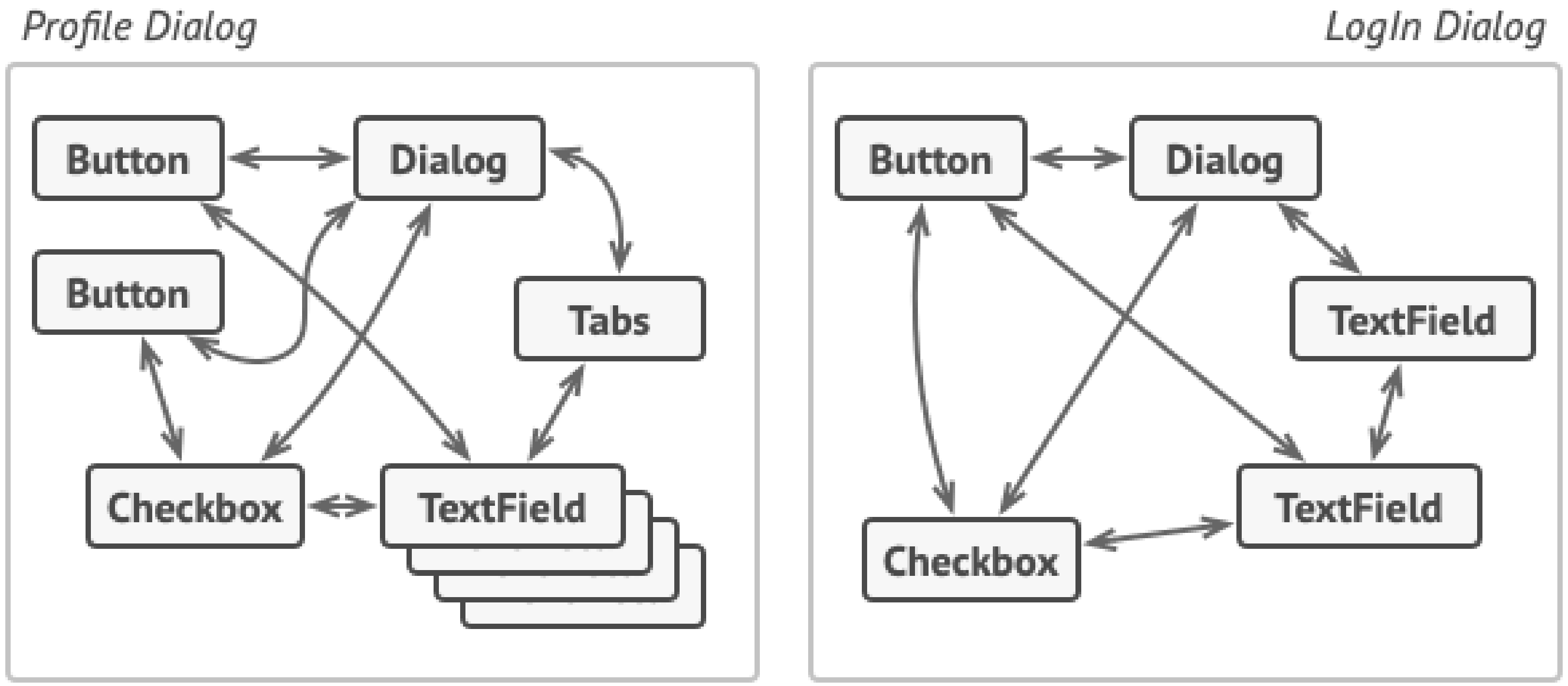
- ✓ Chain of responsibility
- ✓ Command
- ✓ Iterator <<already covered>>
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template method
- ✓ Visitor

Behavioral patterns

- ✓ Chain of responsibility
- ✓ Command
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template method
- ✓ Visitor

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.

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



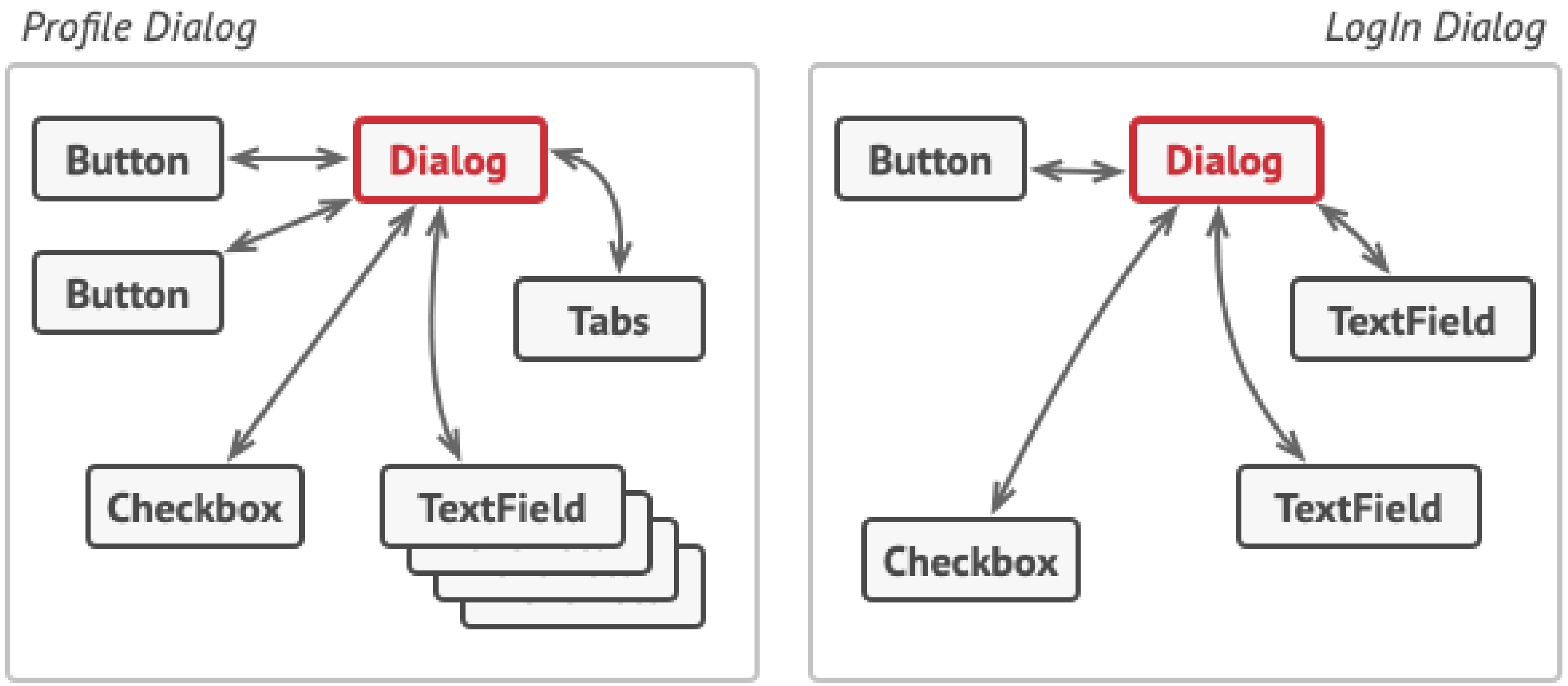
Relations between elements of the user interface can become chaotic as the application evolves.

- 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.

- By having this logic implemented directly inside the code of the form elements you make these elements’ classes much harder to reuse in other forms of the app.
- For example, you won’t be able to use that checkbox class inside another form, because it’s coupled to the dog’s text field. You can use either all the classes involved in rendering the profile form, or none at all.

- 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.



Steps to Implement:

1. **Define a mediator interface** to handle communication.
2. **Create concrete mediator classes** to manage interactions.
3. **Create colleague classes** that depend on the mediator for communication.
4. Colleague objects notify the mediator instead of interacting directly with each other.

A **chat room** where users communicate through a central **ChatMediator**.

Mediator Interface:

```
class ChatMediator:
    def send_message(self, message, sender):
        pass
```

Concrete Mediator

```
class ChatRoom(ChatMediator):
    def __init__(self):
        self.users = []

    def add_user(self, user):
        self.users.append(user)

    def send_message(self, message, sender):
        for user in self.users:
            if user != sender:
                user.receive(message, sender.name)
```

Colleague Class:

```
class User:
    def __init__(self, name, mediator):
        self.name = name
        self.mediator = mediator

    def send(self, message):
        print(f"{self.name} sends: {message}")
        self.mediator.send_message(message, self)

    def receive(self, message, sender_name):
        print(f"{self.name} receives from {sender_name}:
{message}")
```

Client code

```
# Create mediator
chat_room = ChatRoom()

# Create users
user1 = User("Alice", chat_room)
user2 = User("Bob", chat_room)
user3 = User("Charlie", chat_room)

# Add users to the chat room
chat_room.add_user(user1)
chat_room.add_user(user2)
chat_room.add_user(user3)

# Users send messages
user1.send("Hello everyone!")
user2.send("Hi Alice!")
user3.send("Hey Alice and Bob!")
```

- 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

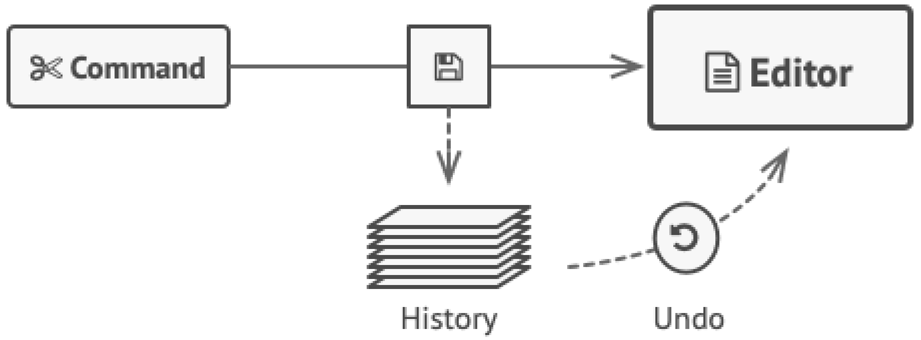
Over time a mediator can evolve into a God Object

Behavioral patterns

- ✓ Chain of responsibility
- ✓ Command
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template method
- ✓ Visitor

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.

- 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.



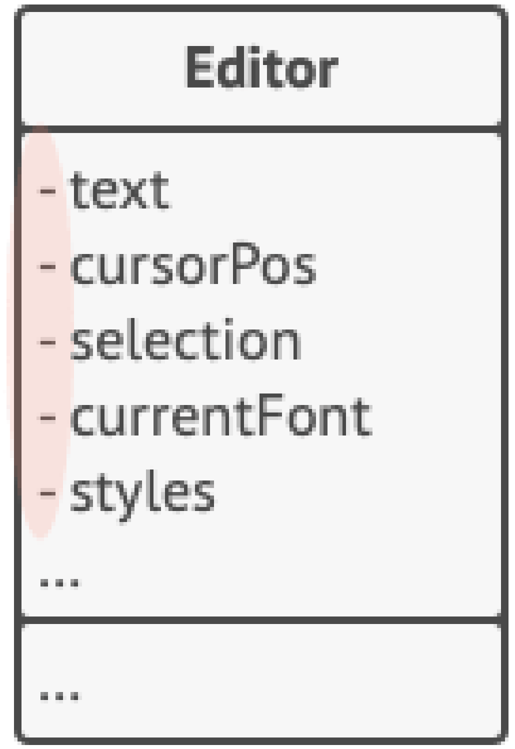
Let’s think about those state snapshots. How exactly would you produce one? You’d probably need to go over all the fields in an object and copy their values into storage. However, this would only work if the object had quite relaxed access restrictions to its contents.

Unfortunately, most real objects won’t let others peek inside them that easily, hiding all significant data in private fields.

In the future, you might decide to refactor some of the editor classes, or add or remove some of the fields. This would also require changing the classes responsible for copying the state of the affected objects.

private = can't copy

public = unsafe



- 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 pattern suggests 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.

Text Editor with Undo Feature

Originator (TextEditor):

```
class TextEditor:
    def __init__(self):
        self.content = ""

    def write(self, text):
        self.content += text

    def save(self):
        return Memento(self.content)

    def restore(self, memento):
        self.content = memento.get_state()

    def __str__(self):
        return self.content
```

Memento (Snapshot of Editor State):

```
class Memento:
    def __init__(self, state):
        self._state = state

    def get_state(self):
        return self._state
```

Caretaker (Manages Mementos):

```
class Caretaker:
    def __init__(self):
        self.mementos = []

    def save(self, memento):
        self.mementos.append(memento)

    def undo(self):
        if not self.mementos:
            return None
        return self.mementos.pop()
```

Client Code:

```
# Create the originator and caretaker
editor = TextEditor()
caretaker = Caretaker()

# Write content and save states
editor.write("Hello, ")
caretaker.save(editor.save())  # Save state 1

editor.write("World!")
caretaker.save(editor.save())  # Save state 2

print("Current Content:", editor)  # Hello, World!

# Undo changes
editor.restore(caretaker.undo())
print("After Undo:", editor)  # Hello,

editor.restore(caretaker.undo())
print("After Undo:", editor)  # (empty)
```

Key Points:

•**Memento** stores snapshots of the object's state.

•**Caretaker** ensures the originator can revert to previous states.

•**Encapsulation**: The internal state of the originator remains hidden from other components.

- 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.
- Most dynamic programming languages, such as PHP, Python and JavaScript, can’t guarantee that the state within the memento stays untouched.

Behavioral patterns

- ✓ Chain of responsibility
- ✓ Command
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template method
- ✓ Visitor

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.

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.

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.

The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.

Following things will be needed:

1) an array field for storing a list of references to subscriber objects and
2) several public methods which allow adding subscribers to and removing them from that list.

Steps to Implement:

1. **Subject Interface**: Define methods for adding, removing, and notifying observers.
2. **Observer Interface**: Define a method for receiving updates.
3. **Concrete Subject**: Implements the subject interface and maintains observer state.
4. **Concrete Observers**: Implement the observer interface and react to updates.

A **WeatherStation** notifies multiple displays (observers) when the temperature changes.

Observer Interface:

```
class Observer:
    def update(self, temperature):
        pass
```

Subject Interface:

```
class Subject:
    def add_observer(self, observer):
        pass

    def remove_observer(self, observer):
        pass

    def notify_observers(self):
        pass
```

Concrete Subject (WeatherStation)

```
class WeatherStation(Subject):
    def __init__(self):
        self.observers = []
        self.temperature = 0

    def add_observer(self, observer):
        self.observers.append(observer)

    def remove_observer(self, observer):
        self.observers.remove(observer)

    def set_temperature(self, temperature):
        self.temperature = temperature
        self.notify_observers()

    def notify_observers(self):
        for observer in self.observers:
            observer.update(self.temperature)
```

Concrete Observers:

```
class PhoneDisplay(Observer):
    def update(self, temperature):
        print(f"Phone Display: Temperature updated to
{temperature}°C")

class WindowDisplay(Observer):
    def update(self, temperature):
        print(f"Window Display: Temperature updated to
{temperature}°C")
```

Client code

```
# Create the subject (WeatherStation)
weather_station = WeatherStation()

# Create observers
phone_display = PhoneDisplay()
window_display = WindowDisplay()

# Add observers to the subject
weather_station.add_observer(phone_display)
weather_station.add_observer(window_display)

# Change temperature
weather_station.set_temperature(25)  # Both observers are
notified
weather_station.set_temperature(30)  # Both observers are
notified

# Remove one observer and change temperature
weather_station.remove_observer(phone_display)
weather_station.set_temperature(20)  # Only WindowDisplay is
notified
```

- **Subject** (WeatherStation): Manages state and notifies observers.
- **Observers** (PhoneDisplay, WindowDisplay): React to changes in the subject.
- **Loose Coupling**: The subject doesn't know the implementation details of the observers.

- 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

Behavioral patterns

- ✓ Chain of responsibility
- ✓ Command
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template method
- ✓ Visitor

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.

- The State pattern is closely related to the concept of a Finite-State Machine.
- There’s a finite number of states which a program can be in. Within any unique state, the program behaves differently, and the program can be switched from one state to another instantaneously.
- However, depending on a current state, the program may or may not switch to certain other states.
- These switching rules, called transitions, are also finite and predetermined.

- Imagine that we have a Document class. A document can be in one of three states: Draft, Moderation and Published. The publish method of the document works a little bit differently in each state:

- ✓ In Draft, it moves the document to moderation.
- ✓ In Moderation, it makes the document public, but only if the current user is an administrator.
- ✓ In Published, it doesn’t do anything at all.

- 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.
- Most methods will contain monstrous conditionals that pick the proper behavior of a method according to the current state.
- Code like this is very difficult to maintain because any change to the transition logic may require changing state conditionals in every method.

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.

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 state classes follow the same interface and the context itself works with these objects through that interface.

Steps to Implement:

1. **Create a State Interface**: Define methods for the behavior.
2. **Create Concrete State Classes**: Implement specific behavior for each state.
3. **Create a Context Class**: Maintain a reference to the current state and delegate behavior to it.

An order can be in different states: **New**, **Processing**, or **Shipped**.

State Interface

```
class OrderState:
    def process(self, order):
        pass

    def ship(self, order):
        pass
```

Concrete State Classes

```
class NewOrderState(OrderState):
    def process(self, order):
        print("Order is now being processed.")
        order.set_state(ProcessingOrderState())

    def ship(self, order):
        print("Order cannot be shipped. It is still new.")

class ProcessingOrderState(OrderState):
    def process(self, order):
        print("Order is already being processed.")

    def ship(self, order):
        print("Order is now shipped.")
        order.set_state(ShippedOrderState())

class ShippedOrderState(OrderState):
    def process(self, order):
        print("Cannot process. Order is already shipped.")

    def ship(self, order):
        print("Order is already shipped.")
```

Context Class

```
class Order:
    def __init__(self):
        self.state = NewOrderState()

    def set_state(self, state):
        self.state = state

    def process(self):
        self.state.process(self)

    def ship(self):
        self.state.ship(self)
```

Client code

```
order = Order()

# Process the order
order.process()  # Order is now being processed.

# Try shipping before processing
order.ship()  # Order is now shipped.

# Try processing after shipping
order.process()  # Cannot process. Order is already shipped.
```

- 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.

Behavioral patterns

- ✓ Chain of responsibility
- ✓ Command
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template method
- ✓ Visitor

Strategy is a behavioral design pattern that enables selecting an algorithm's behavior at runtime. It allows you to define a family of algorithms, encapsulate them in separate classes, and make them interchangeable without altering the client code.

Imagine an application that sorts data differently based on the input size or type (e.g., **QuickSort**, **MergeSort**, or **BubbleSort**).

```
class Sorter:
    def sort(self, data, algorithm):
        if algorithm == "quick":
            print("Sorting with QuickSort")
        elif algorithm == "merge":
            print("Sorting with MergeSort")
        elif algorithm == "bubble":
            print("Sorting with BubbleSort")
        else:
            print("Invalid sorting algorithm")
```

Adding a new algorithm means modifying the sort method, violating the Open/Closed Principle. The class becomes cluttered with if-else statements.

- 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.
- 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

A shopping cart applies different discounts: **No Discount**, **Seasonal Discount**, or **Promotional Discount**.

1. Define Strategy Interface

```
class DiscountStrategy:
    def calculate_discount(self, amount):
        pass
```

2. Create Concrete Strategies

```
class NoDiscount(DiscountStrategy):
    def calculate_discount(self, amount):
        return amount  # No discount applied

class SeasonalDiscount(DiscountStrategy):
    def calculate_discount(self, amount):
        return amount * 0.9  # 10% discount

class PromotionalDiscount(DiscountStrategy):
    def calculate_discount(self, amount):
        return amount * 0.8  # 20% discount
```

3. Context Class

```
class ShoppingCart:
    def __init__(self, discount_strategy):
        self.discount_strategy = discount_strategy

    def set_discount_strategy(self, discount_strategy):
        self.discount_strategy = discount_strategy

    def calculate_total(self, amount):
        return self.discount_strategy.calculate_discount(amount)
```

Client code

```
# No Discount
cart = ShoppingCart(NoDiscount())
print("Total with No Discount:", cart.calculate_total(100))  # 100

# Apply Seasonal Discount
cart.set_discount_strategy(SeasonalDiscount())
print("Total with Seasonal Discount:", cart.calculate_total(100))  # 90

# Apply Promotional Discount
cart.set_discount_strategy(PromotionalDiscount())
print("Total with Promotional Discount:", cart.calculate_total(100))  # 80
```

- 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.

- If you only have a couple of algorithms and they rarely change, there’s no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
- 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. Then you could use these functions exactly as you’d have used the strategy objects, but without bloating your code with extra classes and interfaces.

Behavioral patterns

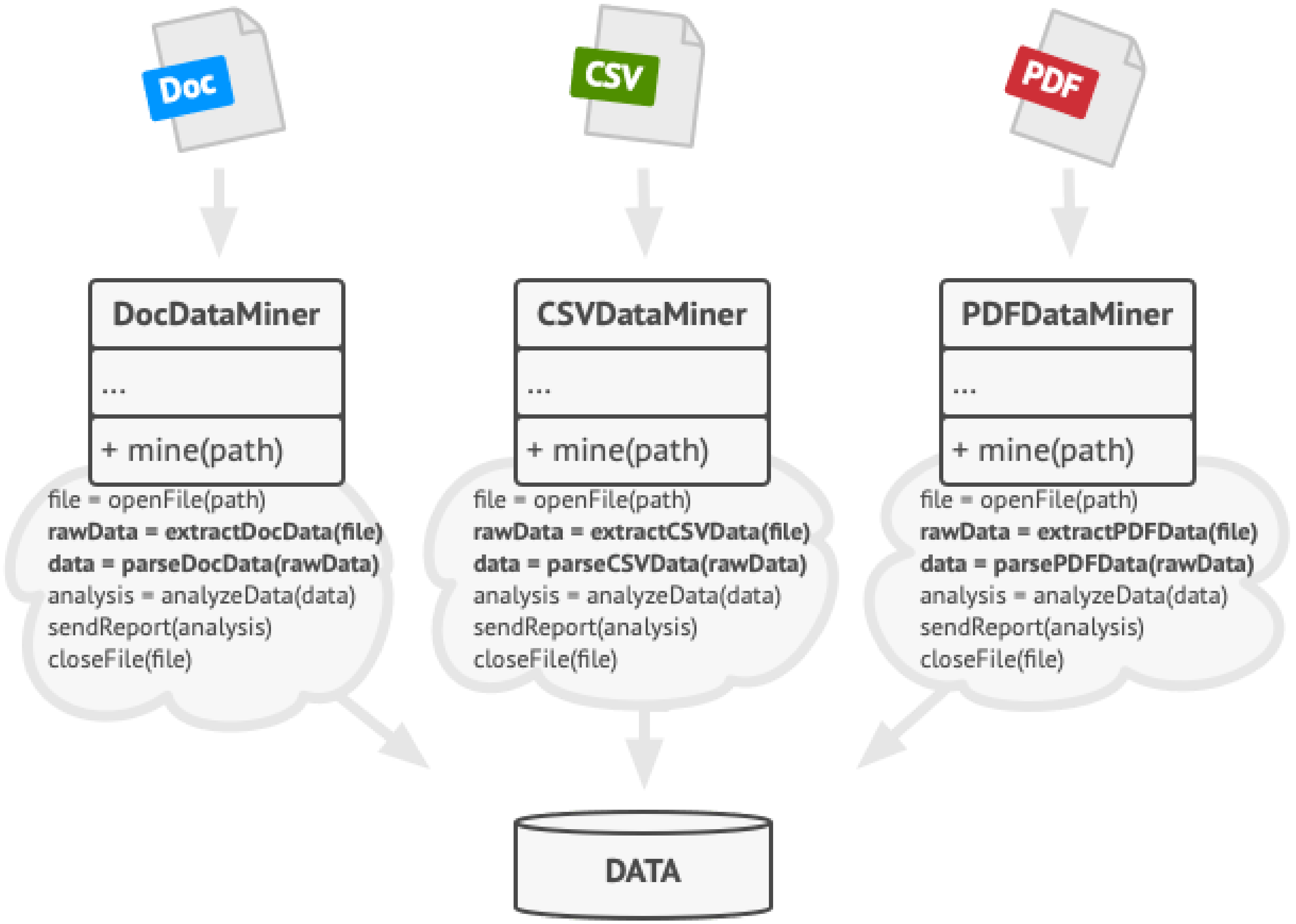
- ✓ Chain of responsibility
- ✓ Command
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template method
- ✓ Visitor

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.

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.

Data mining classes contained a lot of duplicate code.



```
file = openFile(path)
rawData = extractDocData(file)
data = parseDocData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

```
file = openFile(path)
rawData = extractCSVData(file)
data = parseCSVData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

```
file = openFile(path)
rawData = extractPDFData(file)
data = parsePDFData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

- 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?

- 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).

Steps to Implement:

1. **Create an abstract class** with a template method that defines the algorithm's structure.
2. **Implement concrete subclasses** to define specific steps of the algorithm.
3. The template method calls the steps in a fixed order.

An online store processes orders differently based on the payment method: **Credit Card** or **PayPal**.

Abstract Class:

```
from abc import ABC, abstractmethod

class OrderProcessor(ABC):
    def process_order(self):
        self.select_item()
        self.make_payment()
        self.ship_order()

    @abstractmethod
    def select_item(self):
        pass

    @abstractmethod
    def make_payment(self):
        pass

    def ship_order(self):  # Common step
        print("Order shipped to the customer.")
```

Concrete Subclasses:

```
class CreditCardOrderProcessor(OrderProcessor):
    def select_item(self):
        print("Item selected and added to cart.")

    def make_payment(self):
        print("Payment made using Credit Card.")

class PayPalOrderProcessor(OrderProcessor):
    def select_item(self):
        print("Item selected and added to cart.")

    def make_payment(self):
        print("Payment made using PayPal.")
```

```
# Process an order with Credit Card
credit_order = CreditCardOrderProcessor()
credit_order.process_order()

# Process an order with PayPal
paypal_order = PayPalOrderProcessor()
paypal_order.process_order()
```

- 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.

Behavioral patterns

- ✓ Chain of responsibility
- ✓ Command
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template method
- ✓ Visitor

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

Problem it Solves:

- Avoids modifying existing classes when adding new behavior, adhering to the Open/Closed Principle.
- Handles type-dependent operations on a group of related objects.
- Simplifies complex conditional logic (e.g., if-else or switch) for operations based on object types.

Imagine an e-commerce platform where products like **Books** and **Electronics** have different tax rules.

Element Interface

```
class Product:
    def accept(self, visitor):
        pass
```

Concrete Elements:

```
class Book(Product):
    def accept(self, visitor):
        visitor.visit_book(self)

class Electronics(Product):
    def accept(self, visitor):
        visitor.visit_electronics(self)
```

Visitor Interface:

```
class TaxVisitor:
    def visit_book(self, book):
        pass

    def visit_electronics(self, electronics):
        pass
```

Concrete Visitor:

```
class TaxCalculator(TaxVisitor):
    def visit_book(self, book):
        print("Applying 5% tax for books.")

    def visit_electronics(self, electronics):
        print("Applying 15% tax for electronics.")
```

- 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.

- 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.

That’s it