Architectural and Design Patterns
NI-ADP

Quick Overview

Design pattern is a template for solving a certain class of problems. Their purpose is to facilitate the work and make the code easier to read, maintainable, testable, reusable and more efficient, based on best-practices.

Main reasons for Design Patterns

- improved performance
- elimination of a bunch of critical, well-known problems
- security of coding
- best-practices of system design
- cleaner code, readability
- speeds up the development process with known principles
- reduces system complexity, improves cohesion, reduces entanglement
- reusability, testability

Creational Patterns

Object creation and initialization for improved flexibility, reusability and testability.

- **Factory** - create objects with shared interface
- **Abstract Factory** - creates a family of similar objects
- **Builder** - creates complex objects, builds them from steps
- **Prototype** - supports copying of existing objects without dependencies
- **Singleton** - limits the creation to only 1 global instance

Structural Patterns

Code structuralization and decomposition for improved flexibility and efficiency.

- **Adapter** - adapting the interface of a component to fit
- **Bridge** - splitting the interface from the implementations
- **Composite** - uses tree structure for object manipulation
- **Decorator** - dynamically extends functionalities
- **Facade** - defines abstracted interface for easier use
- **Flyweight** - minimizes memory usage by sharing data of similar objects
- **Proxy** - represents an object with another one for less complexity

Behavioral Patterns

Defines the interaction and responsibilities between objects for efficient communication, simplification and comprehensibility.

- **Chain of responsibility** - way of delegating commands into chain of processing
- **Command** - relays command requests to objects
- **Interpreter** - supports the use of language elements in application
- **Mediator** - provides simple communication between objects
- **Memento** - stores and restores of object states
- **Observer** - way of notifying objects about the change of other objects
- **State** - defines how will the behavior of an object change based on other changes
- **Strategy** - includes algorithm to class
- **Visitor** - defines new method on a class without altering it
- **Template Method** - defines a template of operation

Design principles

Principles of good design. Writing good, maintainable, readable, testable, documented, robust, modular, solid code with minimized technical debt.

Technical debt

The cost of additional rework caused by choosing the quicked solution rather than the most effective one. Speedy deliveries and tight deadlines are usually what increase this debt.

Intentional debt - when we make decisions to optimize for the present and not the future

Unintentional debt - when we lack understanding, make mistakes etc.

- Prudent and deliberate debt - when you need to ship it quickly, most common
- Reckless and deliberate debt - when you know how to make it good but decide not to
- Prudent and accidental debt - when you do it good but then find a better way after it's late
- Reckless and accidental debt - when you try to do the best solution without enough knowledge - you make mistakes

The Principles

- **DRY - Don't Repeat Yourself**
    - Duplication makes the code lengthy and wastes a lot of time when it comes to maintaining, debugging or modifying/extending the code. Small changes should be done only in one place.
    - Create common functions or abstract the code to avoid repetition.
- **APO - Avoid Premature Optimization**
    - Don't try to perfect something too early. You might spend a lot of time optimizing something that won't even be used. That leads to wasted resources, increased mistakes and increased code complexity.
    - The first step is to ship something that works. Think about optimization afterwards.
    - "Premature optimization is the root of all evil." - Donald Knuth
- **KISS - Keep It Simple Stupid**
    - Just don't make it complex. We all love simplicity.
    - Make it easy to understand, simplify as much as possible, break the complex parts into smaller chunks, delete what's unnecessary, ...
- **YAGNI - You Ain't Gonna Need It**
    - Build only what you need at the time. Don't waste resources on functionalities that might be useful in the future (they won't)

    - wrong feature built - Cost of build
      right feature built wrong - Cost of delay, Cost of carry, Cost of repair
      right feature built right - Cost of delay
    - By building what's not needed, you increase the cost of basically everything
- **SOLID**
    - **Single Responsibility Principle**
        - Each module has only one responsibility
        - Completely described by its name in one sentence
        - Goal: Changing a module won't affect unrelated behaviors
    - **Open-Closed Principles**
        - Open for extension
        - Closed for modification
        - Goal: To make sure modules are used correctly
    - **Liskov Substitution Principle**
        - A parent can be replaced (substituted) by its child
        - A child should only extend its parent, not modify it
        - Goal: Consistency in using the same type of modules
    - **Interface Segregation Principle**
        - Split (segregate) components into interfaces
        - Clients should only depend on those interfaces that they need
        - Goal: Module executes only what is required, reduces unwanted bugs
    - **Dependency Inversion Principle**
        - High-Level module should not depend on Low-Level module, both should depend on abstraction
            - Don't fuse the module with the tool for its purpose, make it depend on an interface so that any tool that fulfills the interface can be used in its stead
        - Abstraction should not depend on details, details should depend on abstractions
            - The tool needs to only meet the specification of the interface, the module nor interface don't need to know how it actually works

        - Low-Level module - a tool
          High-Level module - uses the tool
          Abstraction - an interface
          Detail - how it works, implementation
        - Goal: Reduces the dependencies by introducing interfaces
- **SoC - Separation of Concerns**
    - At low level - Single Responsibility Principle
    - At high level - Separating architecture into components, layers, tiers (N-Tier structure, data/business/UI)
- **LoD - Law of Demeter**
    - Set of rules to determine who controls and owns what
    - “A friend of a friend is a stranger”
    - Objects should only be able to use directly related objects
        - the same object
        - object passed to me in parameter
        - object created within this method
        - object I own
        - a global object
    - Goal: Keep objects from micromanaging others
- **Tbsr - The boy scout rule**
    - Leave the code in a better state than how I came to it
- **Pola - Principle of least astonishment**
    - It should do what I expected it to do
    - no “wat” effects

Dependency Inversion:

```
// High-level module (business logic)
class BusinessLogic {
    private ServiceInterface service;

    // Constructor injection to provide the dependency
    public BusinessLogic(ServiceInterface service) {
        this.service = service;
    }
```

```
    public void performOperation() {
        // Delegating the operation to the injected service
        service.operation();
    }
}

// Abstraction (interface) representing a service
interface ServiceInterface {
    void operation();
}

// Low-level module implementing the service
class LowLevelService implements ServiceInterface {
    @Override
    public void operation() {
        System.out.println("Performing low-level operation");
    }
}

// Another low-level module implementing a different service
class AnotherLowLevelService implements ServiceInterface {
    @Override
    public void operation() {
        System.out.println("Performing another low-level operation");
    }
}

public class DependencyInversionExample {
    public static void main(String[] args) {
        // High-level module depends on abstraction (ServiceInterface)
        ServiceInterface lowLevelService = new LowLevelService();
        BusinessLogic businessLogic1 = new BusinessLogic(lowLevelService);
        businessLogic1.performOperation();

        // Another low-level module can be easily substituted
        ServiceInterface anotherLowLevelService = new
AnotherLowLevelService();
        BusinessLogic businessLogic2 = new
BusinessLogic(anotherLowLevelService);
        businessLogic2.performOperation();
    }
}
```

Design patterns

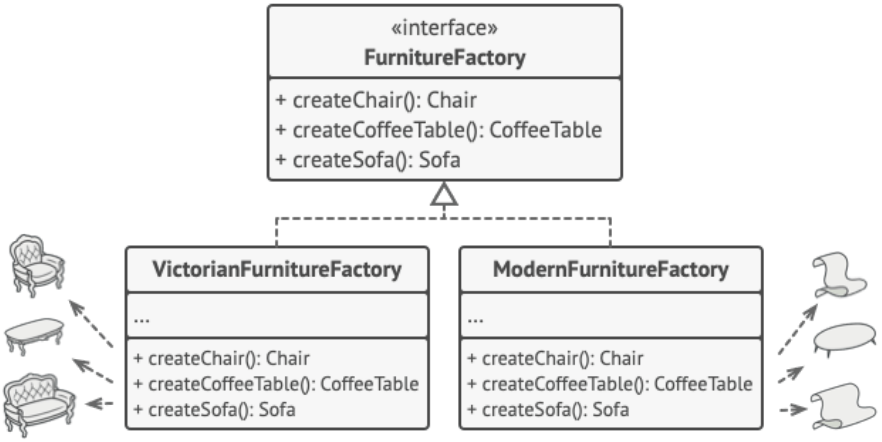
Pre-made blueprints of solutions to commonly occurring problems in SW design. They are high-level descriptions of the solutions while algorithms define a clear set of actions to solve the problem.

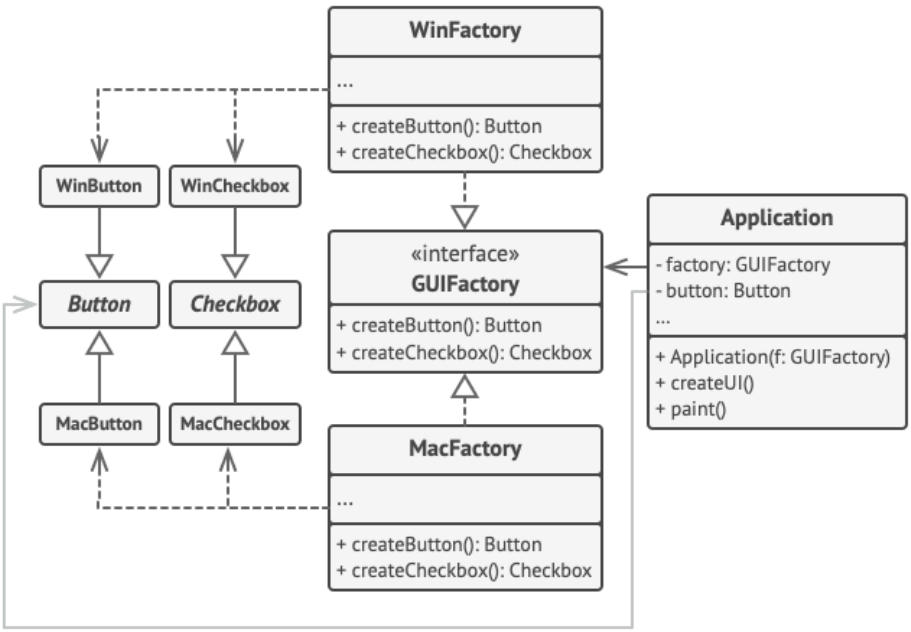
- Intent - briefly describe both the problem and the solution
- Motivation - description of the problem the pattern is intended to solve
- Structure - the structure of modules and their relations to solving the problem

Creational design patterns

Provide object creation mechanisms for increased flexibility and code reusability.

Abstract Factory

- Produce families of related objects
- Problem - A modern-style sofa doesn't match Victorian-style chairs
- 



-
- Pros:
    - The products generated by factories are compatible with each other
    - Avoiding tight coupling between concrete products and client code
    - Single Responsibility Principle - extract product creation in one place
    - Open-Closed Principle - adding new variants without breaking existing code

```
// Abstract product A
interface Button {
    void click();
}

// Concrete product A1
class WindowsButton implements Button {
    @Override
    public void click() {
        System.out.println("Windows button clicked");
    }
}

// Concrete product A2
class MacOSButton implements Button {
    @Override
    public void click() {
        System.out.println("MacOS button clicked");
    }
```

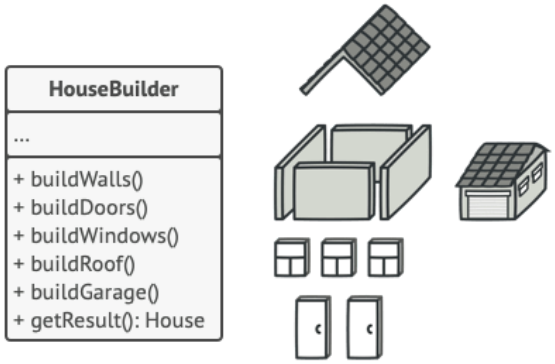
```
}

// Abstract product B
interface CheckBox {
    void check();
}

// Concrete product B1
class WindowsCheckBox implements CheckBox {
    @Override
    public void check() {
        System.out.println("Windows checkbox checked");
    }
}

// Concrete product B2
class MacOSCheckBox implements CheckBox {
    @Override
    public void check() {
        System.out.println("MacOS checkbox checked");
    }
}

// Abstract factory interface
interface GUIFactory {
    Button createButton();
    CheckBox createCheckBox();
}

// Concrete factory for Windows
class WindowsFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new WindowsButton();
    }

    @Override
    public CheckBox createCheckBox() {
        return new WindowsCheckBox();
    }
}
```

```java
// Concrete factory for MacOS
class MacOSFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new MacOSButton();
    }

    @Override
    public CheckBox createCheckBox() {
        return new MacOSCheckBox();
    }
}

// Client code using the abstract factory
public class AbstractFactoryExample {
    public static void main(String[] args) {
        // Create a Windows GUI
        GUIFactory windowsFactory = new WindowsFactory();
        Button windowsButton = windowsFactory.createButton();
        CheckBox windowsCheckBox = windowsFactory.createCheckBox();

        // Use Windows components
        windowsButton.click();
        windowsCheckBox.check();

        // Create a MacOS GUI
        GUIFactory macOSFactory = new MacOSFactory();
        Button macOSButton = macOSFactory.createButton();
        CheckBox macOSCheckBox = macOSFactory.createCheckBox();

        // Use MacOS components
        macOSButton.click();
        macOSCheckBox.check();
    }
}
```

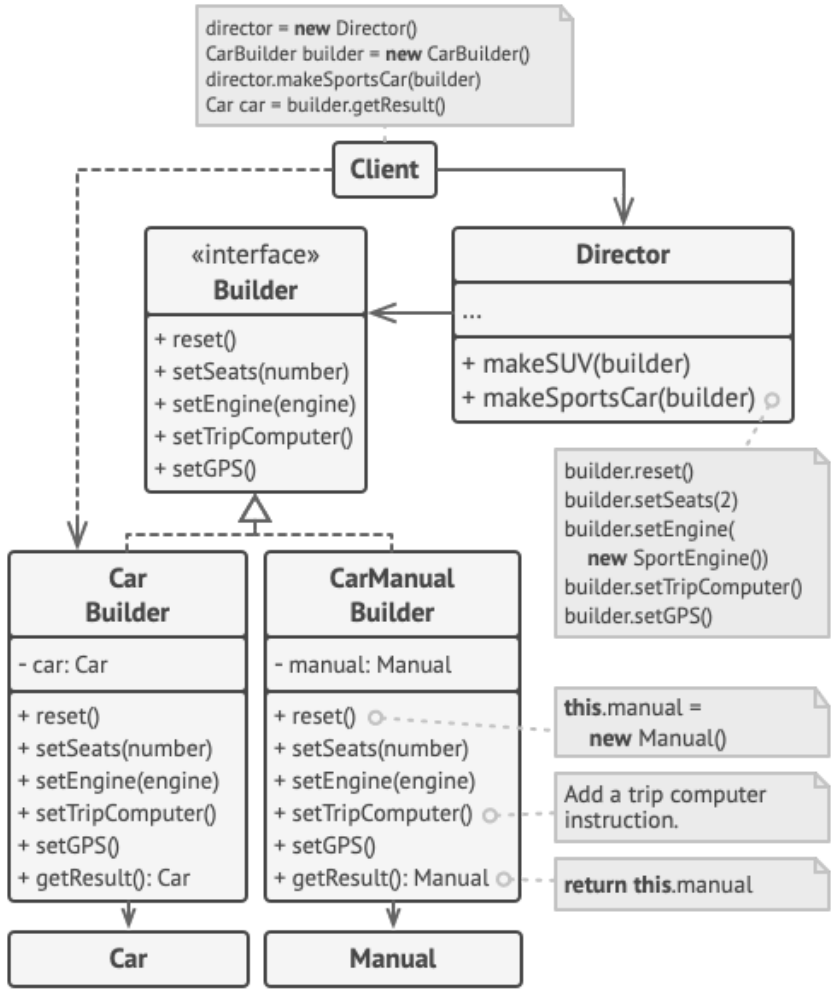
Builder

- Construct complex objects step-by-step

- Problem - various different objects with various components (house vs. house with garage, ...)



-
- Director - a class that can control the order of building



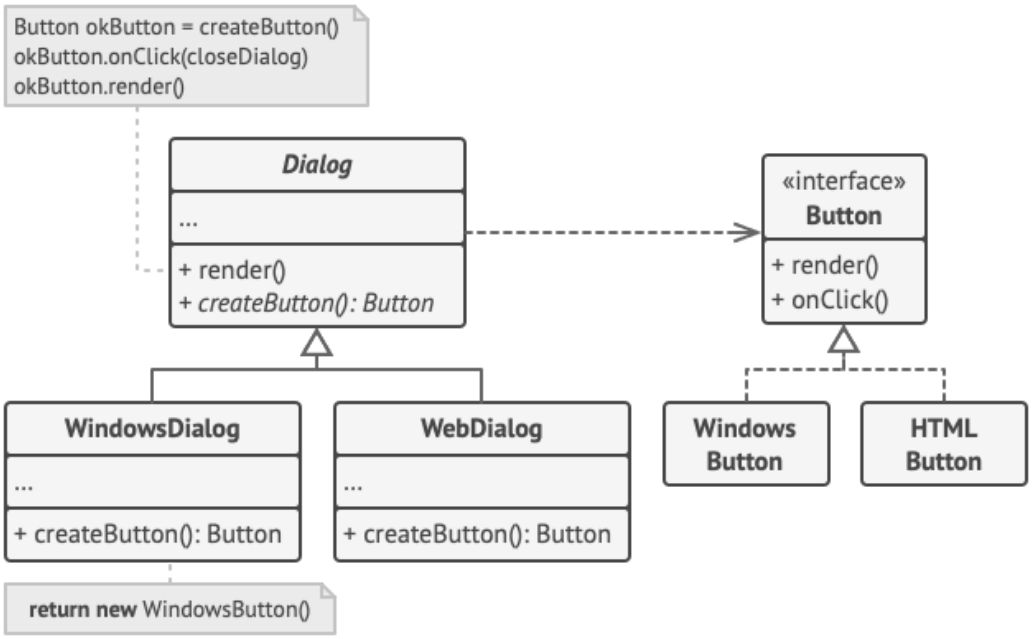
-
- Builder is good alternative to telescoping constructors (10 constructors)
- Pros:

- Reusability of the same construction code for different objects
- Single responsibility principle

```
interface ComputerBuilder {
    ComputerBuilder setCpu(String cpu);
    ComputerBuilder setMemory(String memory);
    ComputerBuilder setStorage(String storage);
    ComputerBuilder setGraphicsCard(String graphicsCard);
    Computer build();
}
...
Computer customDesktop = desktopBuilder
    .setCpu("Intel i9")
    .setMemory("32GB RAM")
    .setStorage("1TB SSD")
    .setGraphicsCard("NVIDIA RTX 3080")
    .build();
```

Factory Method

- Provides an interface for creating objects
- Problem - when you don't know beforehand the exact types of objects you need to be created

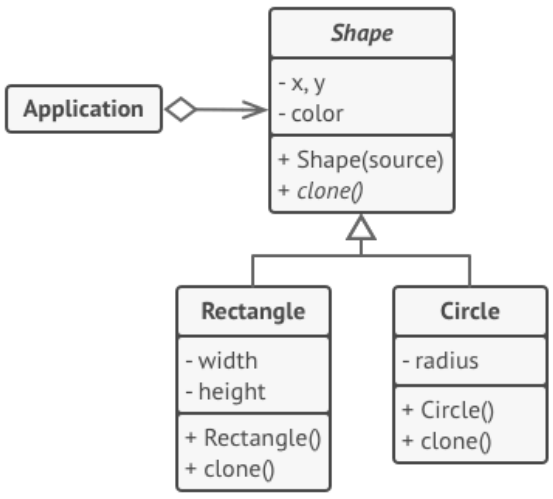


-
- Pros:
    - Avoiding tight coupling between the creator and the created object
    - Single responsibility principle
    - Open-Closed principle - easy adding of new objects

```
// Product interface
interface Product {
    void display();
}

// Concrete product
class ConcreteProduct implements Product {
    @Override
    public void display() {
        System.out.println("Concrete Product");
    }
}

// Creator interface with the factory method
interface Creator {
    Product createProduct();
}

// Concrete creator implementing the factory method
class ConcreteCreator implements Creator {
```

```
    @Override
    public Product createProduct() {
        return new ConcreteProduct();
    }
}

// Client code
public class FactoryMethodExample {
    public static void main(String[] args) {
        // Creating a concrete creator
        Creator creator = new ConcreteCreator();

        // Using the factory method to create a product
        Product product = creator.createProduct();

        // Using the product
        product.display();
    }
}
```

Prototype

- Copying an existing object without the code being dependent on the copy
- Delegate the cloning function to the class itself
- Use prototype when you need to copy something, that's been passed from 3rd party via interface and so on
- 

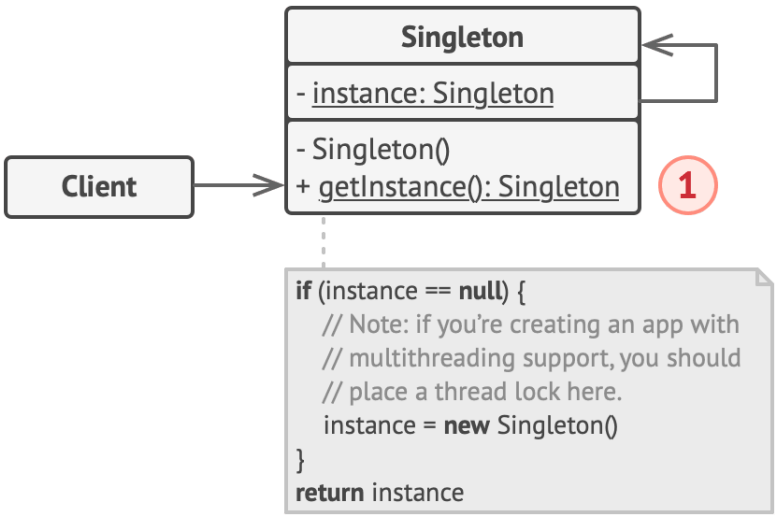
- Pros:
    - we can clone objects without coupling
    - getting rid of initialization
    - producing complex objects more conveniently
    - an alternative to inheritance when dealing with configuration presets for complex objects
- Cons:
    - cloning complex objects with circular references may be problematic

```
// Prototype interface
interface Prototype {
    Prototype clone();
    void setName(String name);
    String getName();
}

// Concrete prototype
class ConcretePrototype implements Prototype {
    private String name;

    public ConcretePrototype(String name) {
        this.name = name;
    }

    @Override
    public Prototype clone() {
        return new ConcretePrototype(this.name);
    }

    @Override
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }
}

// Client code
```

```
public class PrototypeExample {
    public static void main(String[] args) {
        // Creating a prototype
        Prototype original = new ConcretePrototype("Original Object");
        System.out.println("Original Object: " + original.getName());

        // Cloning the prototype to create a new object
        Prototype clone = original.clone();
        System.out.println("Cloned Object: " + clone.getName());

        // Modifying the cloned object
        clone.setName("Modified Object");
        System.out.println("Modified Cloned Object: " + clone.getName());
    }
}
```

Singleton

- When we need to work with only 1 instance from a global access point
- Problem - When we need to control access to some shared source for example
- Clients don't need to know that they are working with the same instance
- It's a safe alternative to global variable



-
- Pros:
    - We know the class has only 1 instance

    - We gain global access point to that instance
    - Object is initialized only when requested for the first time
- Cons:
    - Violates Single Responsibility Principle - solves 2 problems at the same time
    - Can mask bad design
    - Requires special treatment and multithreaded environment
    - Difficult for unit testing (mocking)

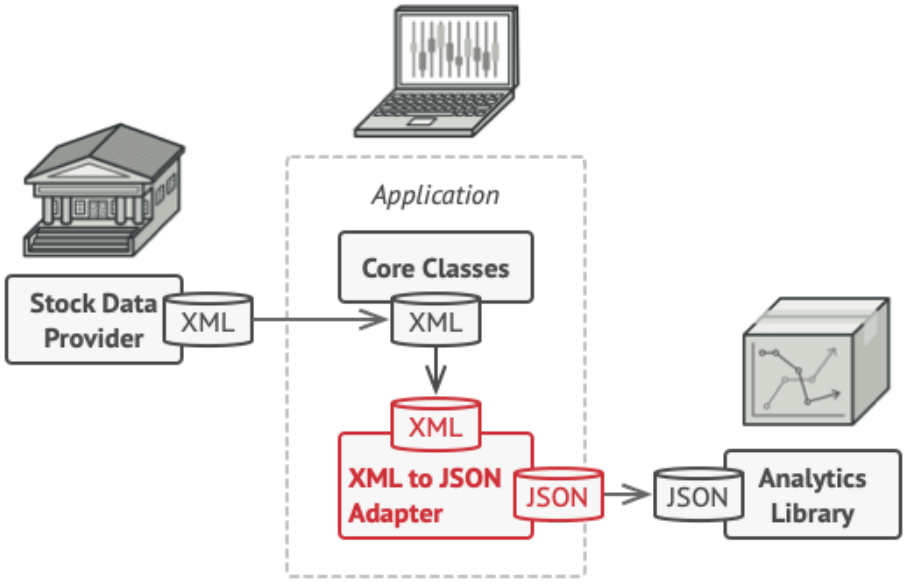
```
public class Singleton {
    // Private static instance variable
    private static Singleton instance;

    // Private constructor to prevent instantiation from outside the class
    private Singleton() {
    }

    // Public method to get the instance of the singleton class
    public static Singleton getInstance() {
        if (instance == null) {
            // Create a new instance only if one does not already exist
            instance = new Singleton();
        }
        return instance;
    }

    // Other methods and properties can be added here
    public void showMessage() {
        System.out.println("Hello, I am a Singleton!");
    }
}

// Client code
public class SingletonExample {
    public static void main(String[] args) {
        // Get the instance of the Singleton class
        Singleton singleton = Singleton.getInstance();

        // Call a method on the Singleton instance
        singleton.showMessage();
    }
}
```

Structural design patterns

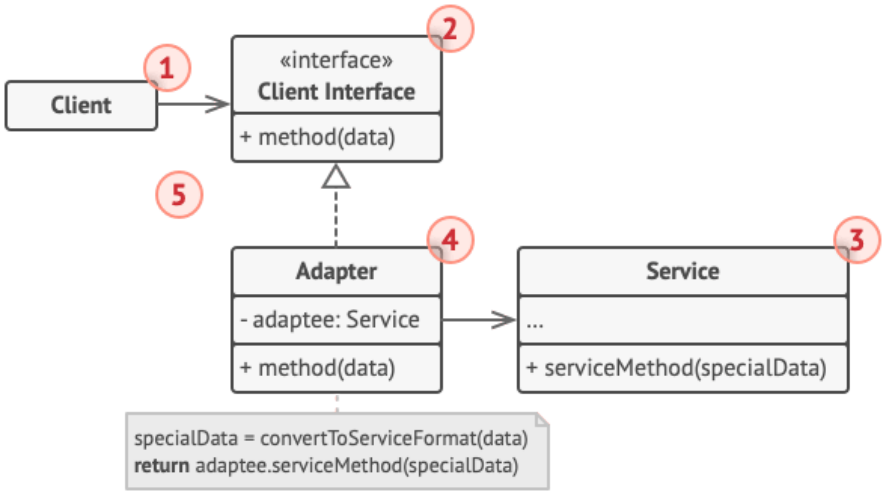
How to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

Adapter

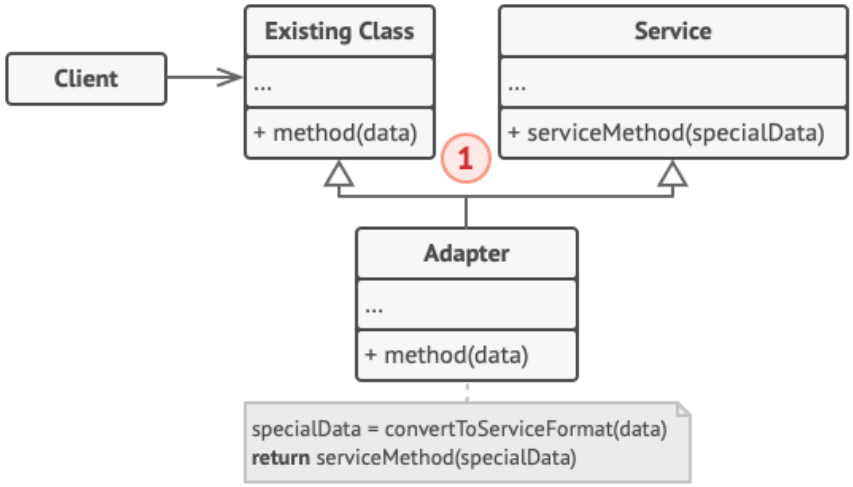
- Allows collaboration between incompatible objects
- Use Adapter when you need to use a class with non-compatible interface with the rest of the code
- Problem: Communication between XML provider and JSON analytics



-
- Object Adapter



- Class Adapter



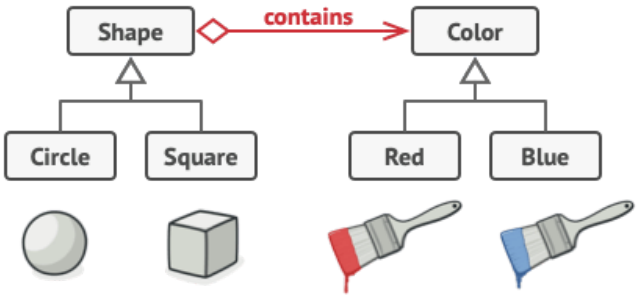
- Pros:
    - Single Responsibility Principle
    - Open-Closed Principle

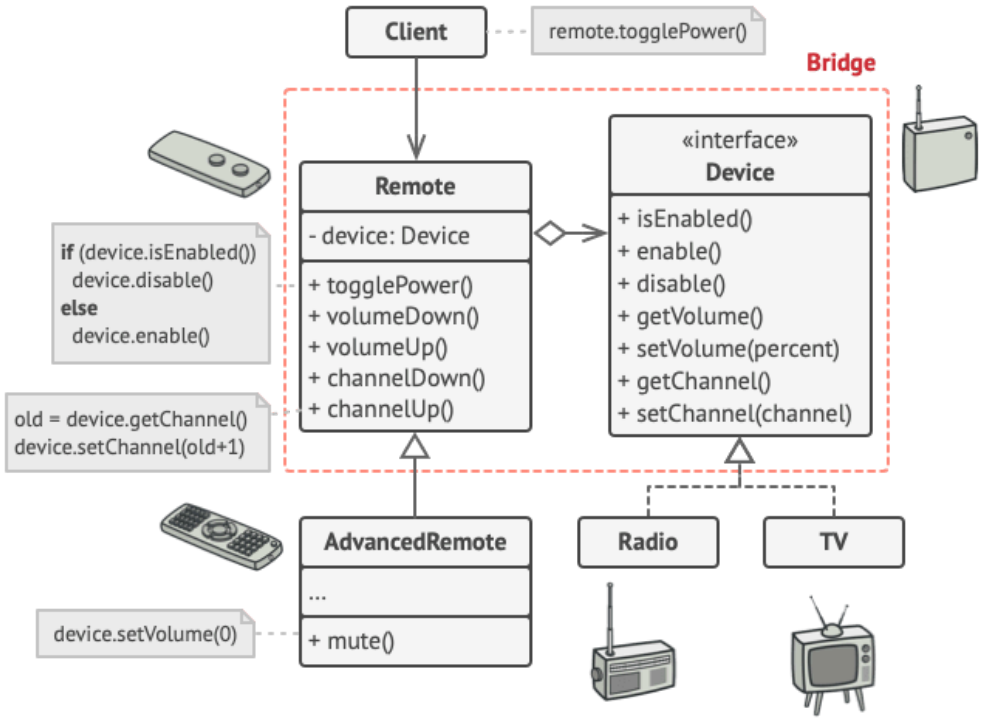
```
// Target interface
interface Target {
    void request();
}

// Adaptee (existing class with an incompatible interface)
class Adaptee {
    void specificRequest() {
        System.out.println("Specific request from Adaptee");
    }
```

```
}

// Adapter class that adapts the Adaptee to the Target interface
class Adapter implements Target {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void request() {
        adaptee.specificRequest();
    }
}

// Client code expecting a Target interface
public class AdapterExample {
    public static void main(String[] args) {
        // Creating an instance of the Adaptee (existing class)
        Adaptee adaptee = new Adaptee();

        // Creating an Adapter to make Adaptee compatible with Target
        Target adapter = new Adapter(adaptee);

        // Using the Target interface to make a request
        adapter.request();
    }
}
```

Bridge

- Lets you split a large module into two separate hierarchies - abstraction and implementation
- Both abstraction and implementation can then be developed independently
- Problem: We have 2 dimensions - shape and color, do we need ShapeColor for every shape and color?
- Bridge switches from inheritance to composition instead

- 

- 

- Pros:
    - create platform-independent classes and apps
    - client code works with high-level abstraction, isn't exposed to platform details
    - Open-Closed principle - new abstractions and implementations independently
    - Single responsibility principle

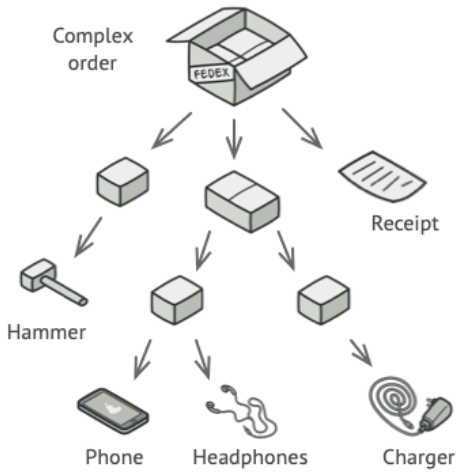
```
// Abstraction interface
interface Shape {
    void draw();
}
```

```java
// Refined Abstraction
class Circle implements Shape {
    private Color color;

    public Circle(Color color) {
        this.color = color;
    }

    @Override
    public void draw() {
        System.out.print("Draw Circle in ");
        color.applyColor();
    }
}

// Implementor interface
interface Color {
    void applyColor();
}

// Concrete Implementor
class RedColor implements Color {
    @Override
    public void applyColor() {
        System.out.println("Red Color");
    }
}

// Another Concrete Implementor
class GreenColor implements Color {
    @Override
    public void applyColor() {
        System.out.println("Green Color");
    }
}

// Client code
public class BridgeExample {
    public static void main(String[] args) {
        // Creating shapes with different colors
        Shape redCircle = new Circle(new RedColor());
        Shape greenCircle = new Circle(new GreenColor());
```

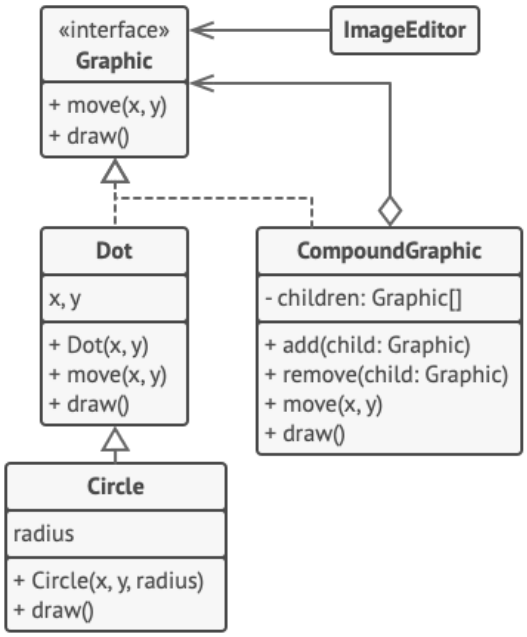
```
        // Drawing shapes
        redCircle.draw();   // Output: Draw Circle in Red Color
        greenCircle.draw(); // Output: Draw Circle in Green Color
    }
}
```

Composite

- Compose objects into tree structures and then work with these trees
- Use Composite only iIf we can represent the model as a tree



-
- Problem: Determine the total price of the whole order above
- Introduce common interface, perform the action recursively



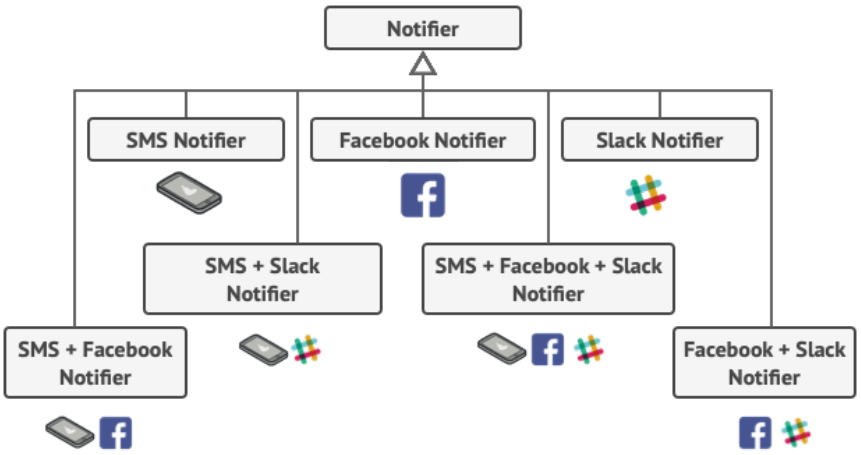
- Pros:
    - Efficient work with complex tree structures, polymorphism, recursion
    - Open-Closed Principle - adding new elements without breaking the code
- Cons:
    - Difficult to provide a common interface for classes that differ too much

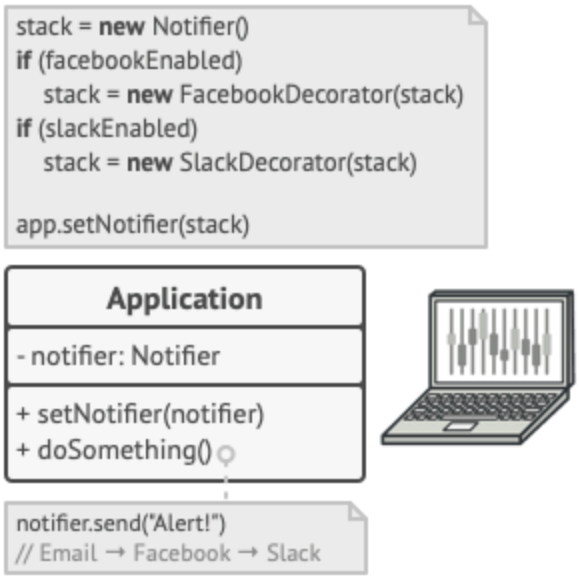
```
// Component interface
interface Graphic {
    void draw();
}

// Leaf class implementing the Component interface
class Circle implements Graphic {
    @Override
    public void draw() {
        System.out.println("Draw Circle");
    }
}

// Leaf class implementing the Component interface
class Square implements Graphic {
    @Override
    public void draw() {
```

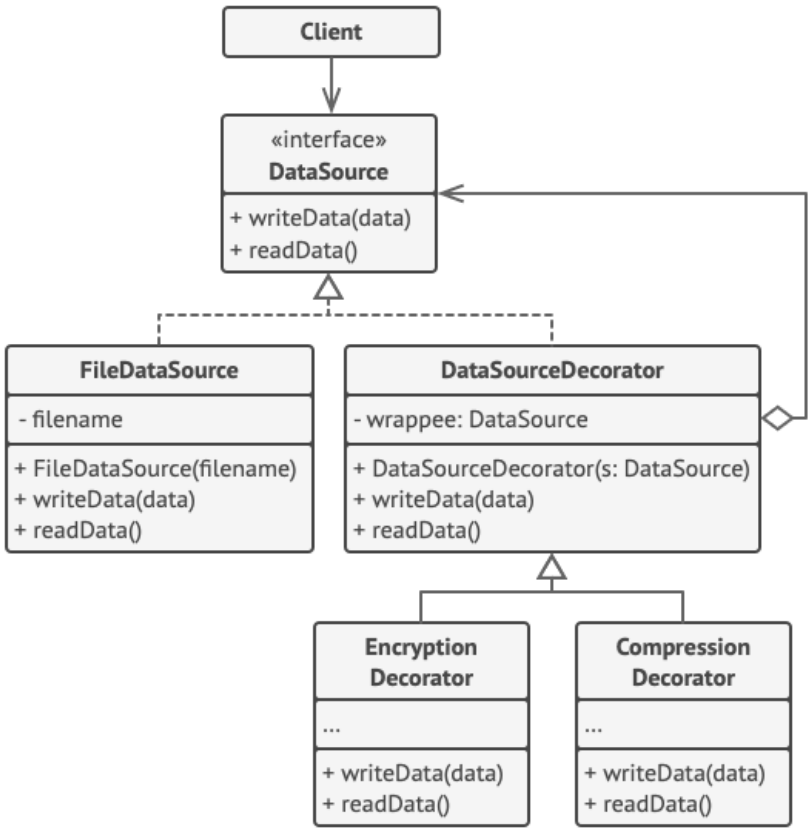
```
        System.out.println("Draw Square");
    }
}

// Composite class implementing the Component interface
class CompositeGraphic implements Graphic {
    private List<Graphic> graphics = new ArrayList<>();

    // Add a graphic to the composite
    public void addGraphic(Graphic graphic) {
        graphics.add(graphic);
    }

    @Override
    public void draw() {
        System.out.println("Composite Graphic:");
        for (Graphic graphic : graphics) {
            graphic.draw();
        }
    }
}

// Client code
public class CompositeExample {
    public static void main(String[] args) {
        // Creating leaf graphics
        Circle circle = new Circle();
        Square square = new Square();

        // Creating a composite graphic
        CompositeGraphic composite = new CompositeGraphic();
        composite.addGraphic(circle);
        composite.addGraphic(square);

        // Drawing individual graphics
        circle.draw();  // Output: Draw Circle
        square.draw();  // Output: Draw Square

        // Drawing composite graphic (which includes the leaf graphics)
        composite.draw();
        /*
         Output:
```

```
        Composite Graphic:
        Draw Circle
        Draw Square
        */
    }
}
```

Decorator (Wrapper)

- Add new behaviors to objects by wrapping them
- Problem: When we want our little Notifier to do more various things and inheritance is a no
- 

- 



- Pros:
    - We can extend objects behavior without making new subclasses
    - We can add/remove responsibilities from an object at runtime
    - We can combine behaviors by wrapping them into multiple decorators
    - Single Responsibility Principle
- Cons:
    - It's hard to remove specific wrapper from wrapper stack
    - It's hard to implement a decorator where the behavior doesn't depend on the decorator stack
    - Initial configuration code of layers might look ugly

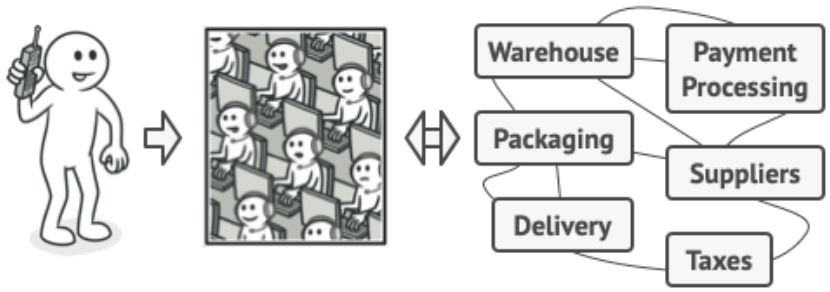
```
// Component interface
interface Text {
    String getContent();
}

// Concrete component
class SimpleText implements Text {
```

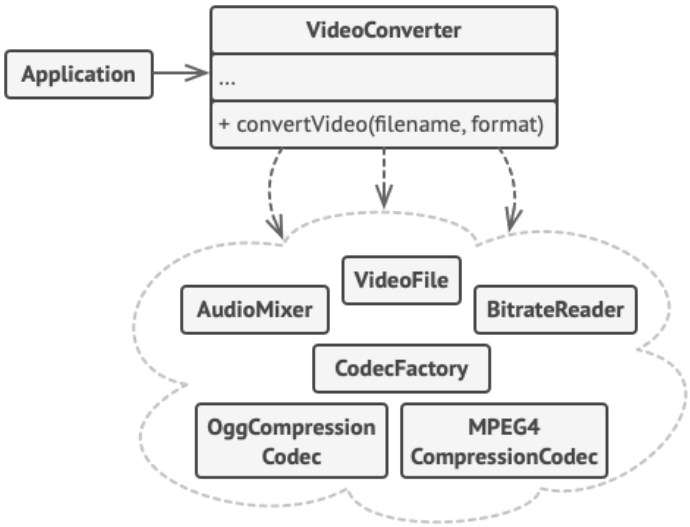
```
    private String content;

    public SimpleText(String content) {
        this.content = content;
    }

    @Override
    public String getContent() {
        return content;
    }
}

// Decorator abstract class
abstract class TextDecorator implements Text {
    protected Text decoratedText;

    public TextDecorator(Text decoratedText) {
        this.decoratedText = decoratedText;
    }

    @Override
    public String getContent() {
        return decoratedText.getContent();
    }
}

// Concrete decorator 1
class BoldTextDecorator extends TextDecorator {
    public BoldTextDecorator(Text decoratedText) {
        super(decoratedText);
    }

    @Override
    public String getContent() {
        return "<b>" + super.getContent() + "</b>";
    }
}

// Concrete decorator 2
class ItalicTextDecorator extends TextDecorator {
    public ItalicTextDecorator(Text decoratedText) {
        super(decoratedText);
```

```
    }

    @Override
    public String getContent() {
        return "<i>" + super.getContent() + "</i>";
    }
}
```

Facade

- Provide simplified interface to a complex structure
- Problem: Our code works with sophisticated processes, libraries and frameworks etc...
- We can abstract all this complexity behind a facade with simplified interface





- Pros:

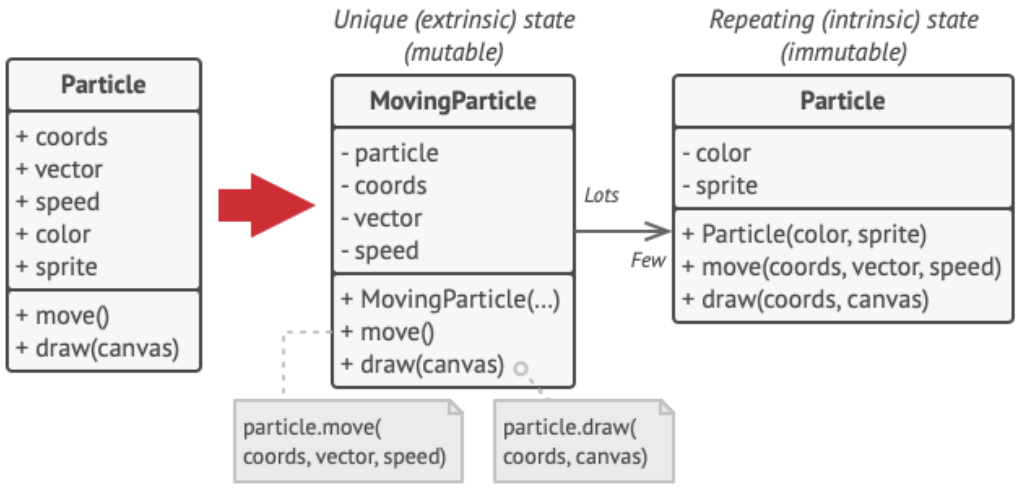
    - isolate code from the complexity of a subsystem
- Cons:
    - A facade can become a God object

```
class InventorySystem {
    public void checkInventory(String product) {
        System.out.println("Checking inventory for product: " + product);
        // Implementation details for checking inventory
    }
}

class PaymentSystem {
    public void processPayment(String product, double amount) {
        System.out.println("Processing payment for product: " + product +
", Amount: " + amount);
        // Implementation details for processing payment
    }
}

class ShippingSystem {
    public void shipProduct(String product, String address) {
        System.out.println("Shipping product: " + product + " to address: "
+ address);
        // Implementation details for shipping
    }
}

// Facade class

class OrderProcessingFacade {
    private InventorySystem inventorySystem;
    private PaymentSystem paymentSystem;
    private ShippingSystem shippingSystem;

    public OrderProcessingFacade() {
        this.inventorySystem = new InventorySystem();
        this.paymentSystem = new PaymentSystem();
        this.shippingSystem = new ShippingSystem();
    }

    public void processOrder(String product, double amount, String address)
{
```

```
        inventorySystem.checkInventory(product);
        paymentSystem.processPayment(product, amount);
        shippingSystem.shipProduct(product, address);
        System.out.println("Order processed successfully for product: " +
product);
    }
}
```

Flyweight

- Fit more objects into memory by sharing common parts of state between them instead of keeping them in each object
- Problem: We want to show a cool particle effect, each particle drains memory
- Use Flyweight only if we need to optimize memory resources



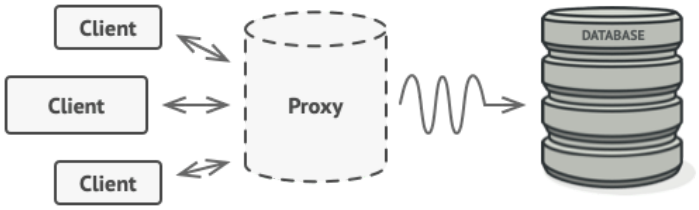
-
- Pros:
    - we can save memory
- Cons:
    - RAM/CPU tradeoff
    - The separation of an object might look weird and make no sense, needs additional comments to explain the purpose

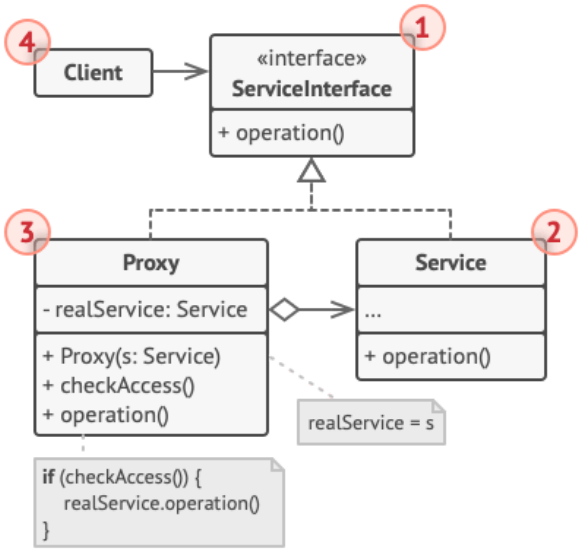
```
// Flyweight interface
interface TextCharacter {
    void draw(String font);
```

```
}

// Concrete Flyweight
class ConcreteTextCharacter implements TextCharacter {
    private char character;

    public ConcreteTextCharacter(char character) {
        this.character = character;
    }

    @Override
    public void draw(String font) {
        System.out.println("Character: " + character + ", Font: " + font);
    }
}

// Flyweight Factory
class TextCharacterFactory {
    private Map<Character, TextCharacter> characterMap = new HashMap<>();

    public TextCharacter getCharacter(char key) {
        // If the character exists in the map, return it; otherwise, create
a new one and store it in the map
        return characterMap.computeIfAbsent(key,
ConcreteTextCharacter::new);
    }
}

// Client code
public class FlyweightExample {
    public static void main(String[] args) {
        TextCharacterFactory characterFactory = new TextCharacterFactory();

        // Client code representing a sequence of characters
        String text = "Hello, Flyweight Pattern!";
        for (char c : text.toCharArray()) {
            TextCharacter character = characterFactory.getCharacter(c);
            // Assume that the font is different for even and odd
characters
            character.draw((c % 2 == 0) ? "Arial" : "Times New Roman");
        }
    }
```

```
}
```

Proxy

- Provide substitute for another object - Proxy controls the original object, allows us to perform action before/after it goes through the original object
- Problem: An object drains large amounts of system resources, so we provide a proxy that will initialize it only when it's really needed - the object may come from 3rd party etc.
- 

- The proxy implements the same interface, allowing us to perform additional actions before/after, such as lazy initialization, monitoring, caching, access control ...
- 

- Pros:
  - We can control the service object without clients knowing
  - We can manage the lifecycle of the service object
  - Proxy works even if the service object doesn't

    - Open-Closed Principle - adding new proxies
    - Performance optimization - caching
- Cons:
    - Response might get delayed

```
interface File {
    void display();
}

// RealSubject
class RealFile implements File {
    private String filename;

    public RealFile(String filename) {
        this.filename = filename;
        loadFileFromDisk();
    }

    private void loadFileFromDisk() {
        System.out.println("Loading file from disk: " + filename);
    }

    @Override
    public void display() {
        System.out.println("Displaying file: " + filename);
    }
}

// Proxy
class FileProxy implements File {
    private RealFile realFile;
    private String filename;

    public FileProxy(String filename) {
        this.filename = filename;
    }

    @Override
    public void display() {
        if (realFile == null) {
            realFile = new RealFile(filename);
        }
```

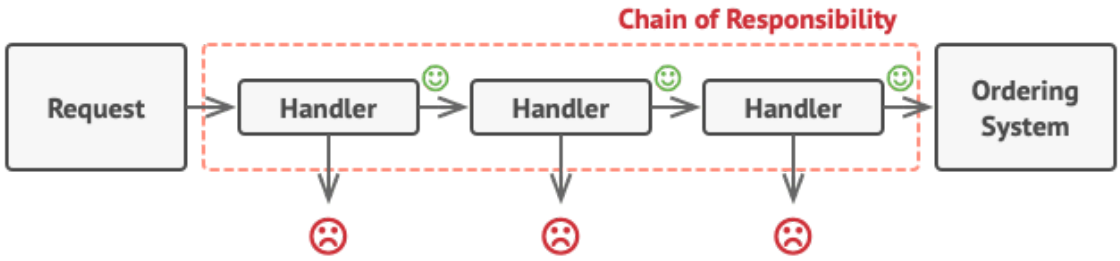
```
        realFile.display();
    }
}
```

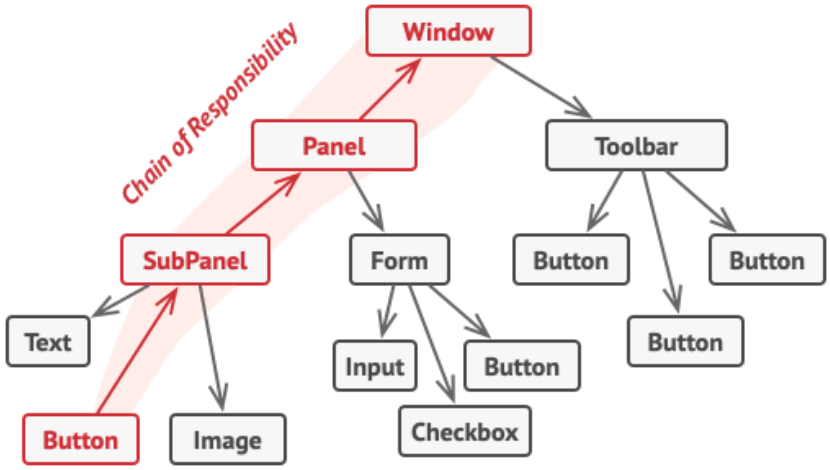
Behavioral design patterns

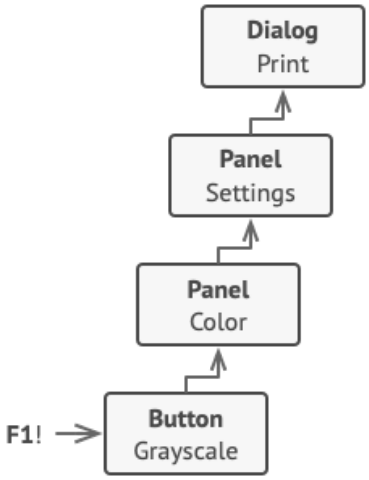
Patterns concerned with algorithms and assignment of responsibilities between objects.

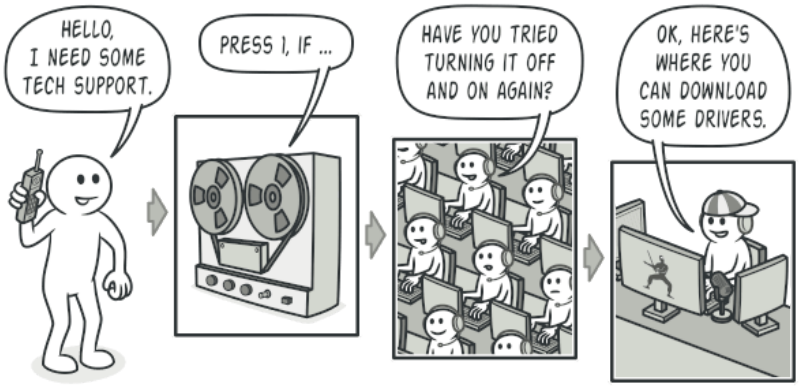
Chain of responsibility

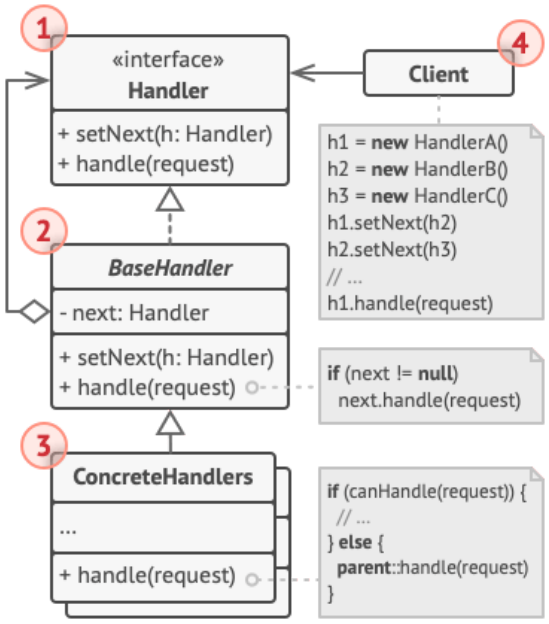
- Pass requests along a chain of handlers. Each handler decides whether to process it or pass it next
- Problem: Authentication, Authorization, Validation etc before processing an order
- Use Chain of responsibility when it's expected to process different kind of requests in various ways



- 
Chain of Responsibility
Window
Panel
Toolbar
SubPanel
Form
Button
Button
Text
Input
Button
Button
Button
Image
Checkbox

- 
Dialog
Print
Panel
Settings
Panel
Color
F1!
Button
Grayscale

- 
HELLO, I NEED SOME TECH SUPPORT.
PRESS 1, IF ...
HAVE YOU TRIED TURNING IT OFF AND ON AGAIN?
OK, HERE'S WHERE YOU CAN DOWNLOAD SOME DRIVERS.



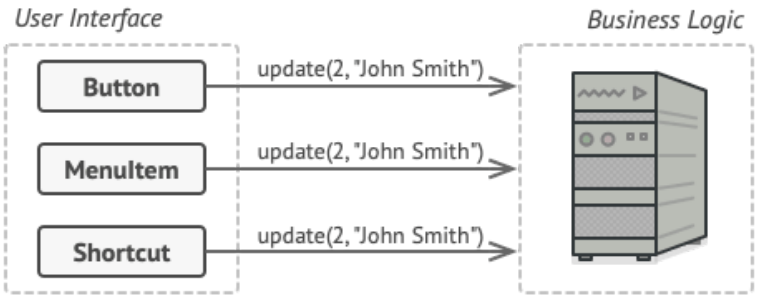
- Pros:
    - We can control the order of request handling
    - Single Responsibility Principle
    - Open-Closed Principle - adding new handlers

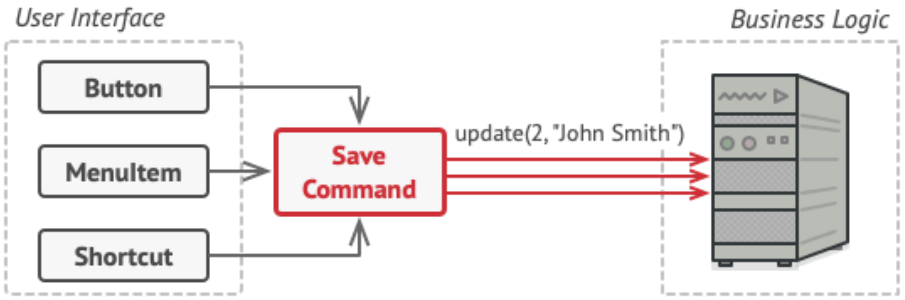
```
// Handler interface
interface RequestHandler {
    void handleRequest(Request request);
}

// Concrete Handler 1
class ConcreteHandler1 implements RequestHandler {
    private static final int THRESHOLD = 10;
    private RequestHandler nextHandler;

    public void setNextHandler(RequestHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    @Override
    public void handleRequest(Request request) {
        if (request.getPriority() <= THRESHOLD) {
            System.out.println("ConcreteHandler1 handling request with
priority " + request.getPriority());
        } else if (nextHandler != null) {
```

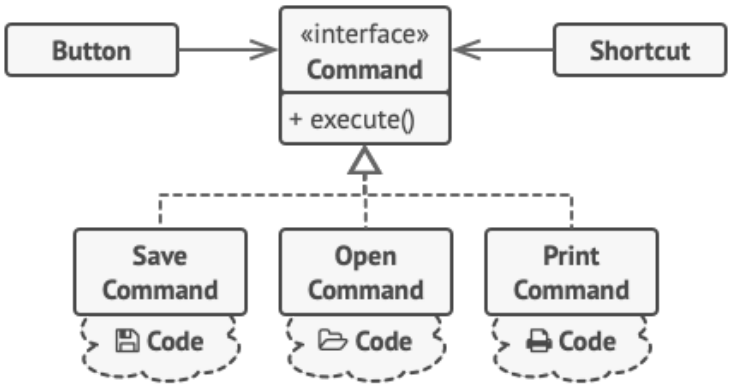
```
            nextHandler.handleRequest(request);
        }
    }
}

// Concrete Handler 2
class ConcreteHandler2 implements RequestHandler {
    private static final int THRESHOLD = 20;

    @Override
    public void handleRequest(Request request) {
        if (request.getPriority() <= THRESHOLD) {
            System.out.println("ConcreteHandler2 handling request with
priority " + request.getPriority());
        } else {
            System.out.println("Request cannot be handled by any
handler.");
        }
    }
}

// Request class
class Request {
    private int priority;

    public Request(int priority) {
        this.priority = priority;
    }

    public int getPriority() {
        return priority;
    }
}

// Client code
public class ChainOfResponsibilityExample {
    public static void main(String[] args) {
        // Create handlers
        RequestHandler handler1 = new ConcreteHandler1();
        RequestHandler handler2 = new ConcreteHandler2();

        // Set up the chain
```

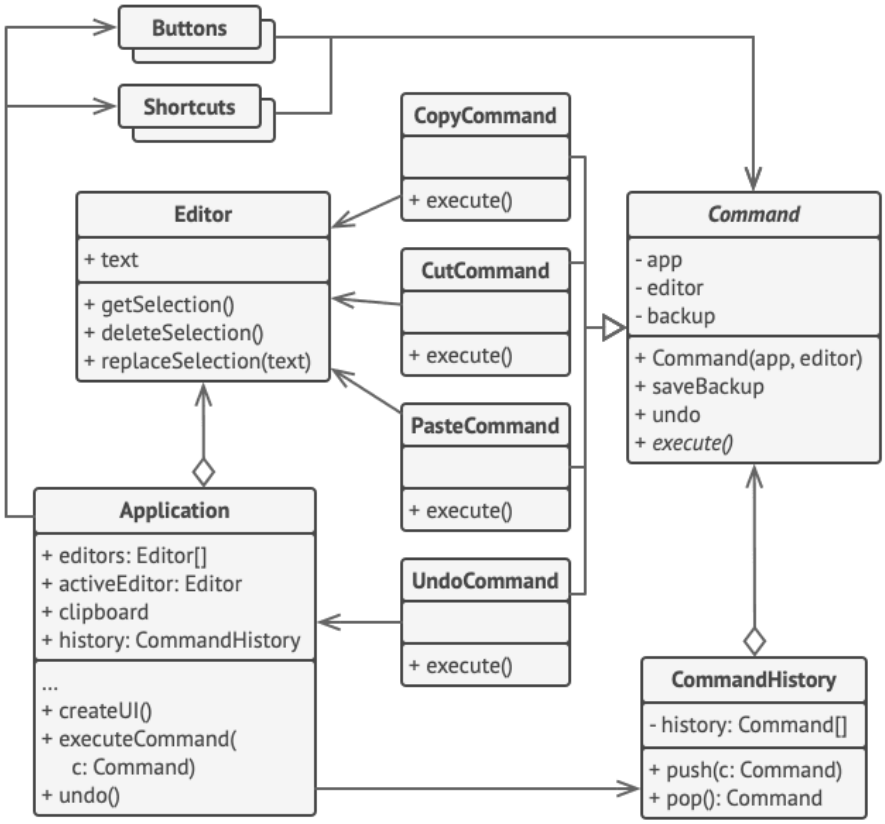
```
        handler1.setNextHandler(handler2);

        // Create requests
        Request request1 = new Request(5);
        Request request2 = new Request(15);

        // Process requests through the chain
        handler1.handleRequest(request1);
        handler1.handleRequest(request2);
    }
}
```

Command

- Turn a whole request into an object that can be manipulated with, delayed, queued or executed
- Problem: We have a bunch of different buttons doing different things. Where to put their handlers?
- 

- 





- Pros:
    - Single Responsibility Principle
    - Open-Closed Principle - adding new commands
    - We can implement Undo/Redo
    - We can assemble multiple commands into a complex one

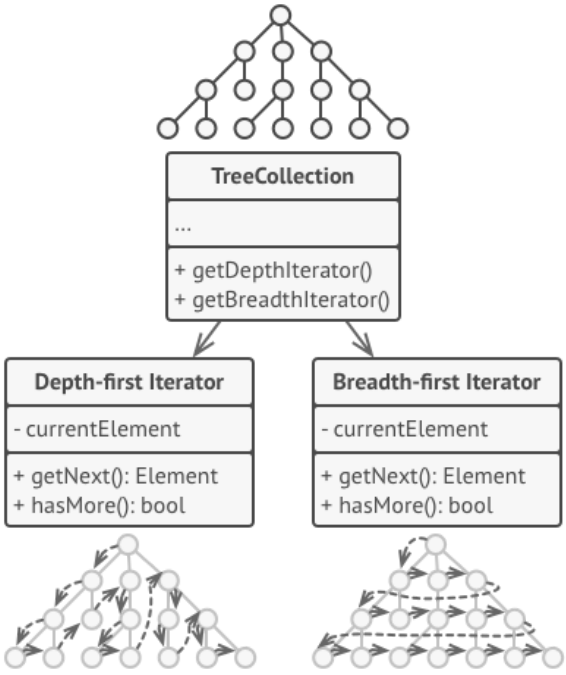
```
// Command interface
interface Command {
    void execute();
```

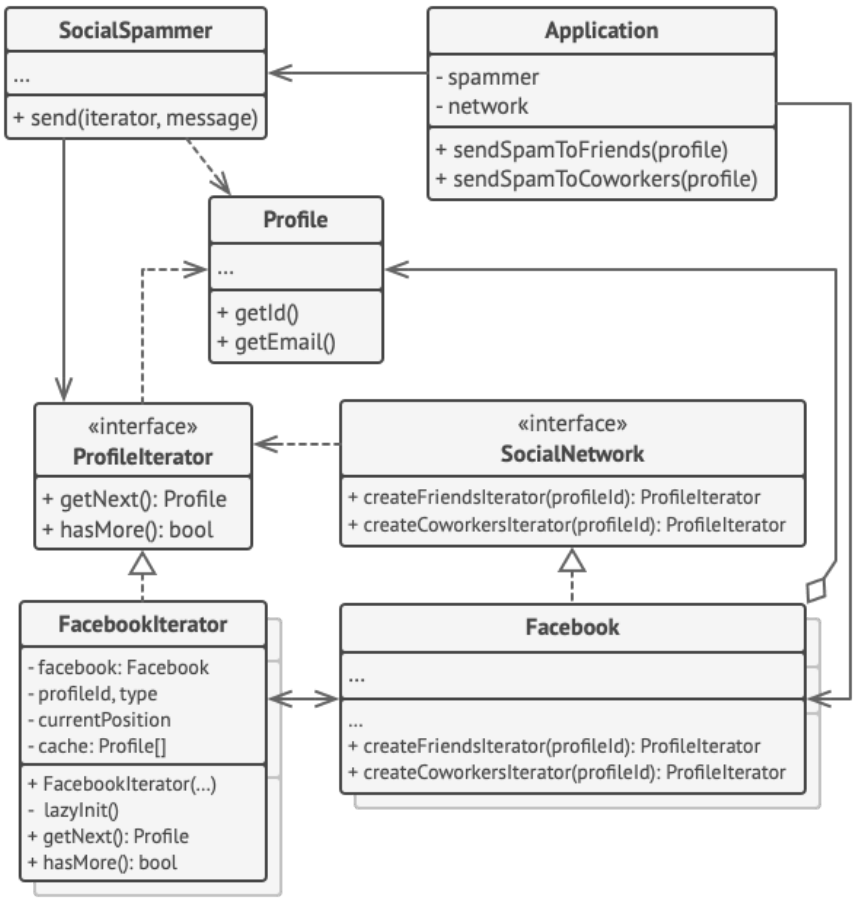
```
}

// Concrete Command 1
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }
}

// Concrete Command 2
class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOff();
    }
}

// Receiver
class Light {
    private String location;

    public Light(String location) {
        this.location = location;
    }

    public void turnOn() {
        System.out.println(location + " Light is ON");
    }
```

```
    public void turnOff() {
        System.out.println(location + " Light is OFF");
    }
}

// Invoker
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

Iterator

- Traversing a collection without exposing its underlying representation (list, stack, tree, ...)
- Most programming languages have various Iterators already implemented
- Problem: We have various types of collections - trees, arrays, lists, stacks, matrices, ... and we want to iterate through them

- 
TreeCollection
...
+ getDepthIterator()
+ getBreadthIterator()
Depth-first Iterator
- currentElement
+ getNext(): Element
+ hasMore(): bool
Breadth-first Iterator
- currentElement
+ getNext(): Element
+ hasMore(): bool


SocialSpammer
...
+ send(iterator, message)
Application
- spammer
- network
+ sendSpamToFriends(profile)
+ sendSpamToCoworkers(profile)
Profile
...
+ getId()
+ getEmail()
«interface»
ProfileIterator
+ getNext(): Profile
+ hasMore(): bool
«interface»
SocialNetwork
+ createFriendsIterator(profileId): ProfileIterator
+ createCoworkersIterator(profileId): ProfileIterator
FacebookIterator
- facebook: Facebook
- profileId, type
- currentPosition
- cache: Profile[]
+ FacebookIterator(...)
- lazyInit()
+ getNext(): Profile
+ hasMore(): bool
Facebook
...
...
+ createFriendsIterator(profileId): ProfileIterator
+ createCoworkersIterator(profileId): ProfileIterator

- Pros:

    - Single Responsibility Principle
    - Open-Closed Principle - adding new types of collections and iterators
    - We can iterate through the same collection in parallel
    - We can pause the iteration and continue later
- Cons:
    - This pattern can be an overkill for simple collections
    - Iterator may be less efficient than a loop

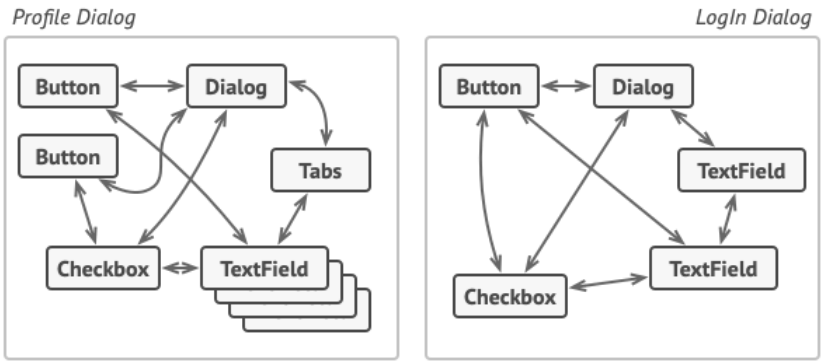
```
// Aggregate interface
interface BookCollection {
    Iterator<Book> iterator();
}

// Concrete Aggregate
class Library implements BookCollection {
    private List<Book> books;

    public Library() {
        this.books = new ArrayList<>();
    }

    public void addBook(Book book) {
        books.add(book);
    }

    @Override
    public Iterator<Book> iterator() {
        return new LibraryIterator(books);
    }
}

// Iterator interface
interface Iterator<T> {
    boolean hasNext();
    T next();
}

// Concrete Iterator
class LibraryIterator implements Iterator<Book> {
    private List<Book> books;
    private int currentIndex;
```

```
    public LibraryIterator(List<Book> books) {
        this.books = books;
        this.currentIndex = 0;
    }

    @Override
    public boolean hasNext() {
        return currentIndex < books.size();
    }

    @Override
    public Book next() {
        if (hasNext()) {
            Book nextBook = books.get(currentIndex);
            currentIndex++;
            return nextBook;
        } else {
            throw new UnsupportedOperationException("No more books in the
collection.");
        }
    }
}

// Book class
class Book {
    private String title;

    public Book(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }
}

// Client code
public class IteratorExample {
    public static void main(String[] args) {
        // Create a library and add books
        Library library = new Library();
```

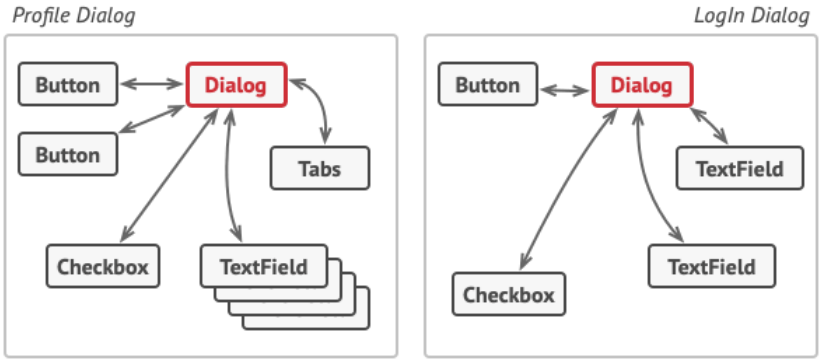
```
        library.addBook(new Book("Design Patterns: Elements of Reusable
Object-Oriented Software"));
        library.addBook(new Book("Clean Code: A Handbook of Agile Software
Craftsmanship"));
        library.addBook(new Book("Effective Java"));

        // Use the iterator to traverse the books
        Iterator<Book> iterator = library.iterator();
        while (iterator.hasNext()) {
            Book book = iterator.next();
            System.out.println("Book Title: " + book.getTitle());
        }
    }
}
```

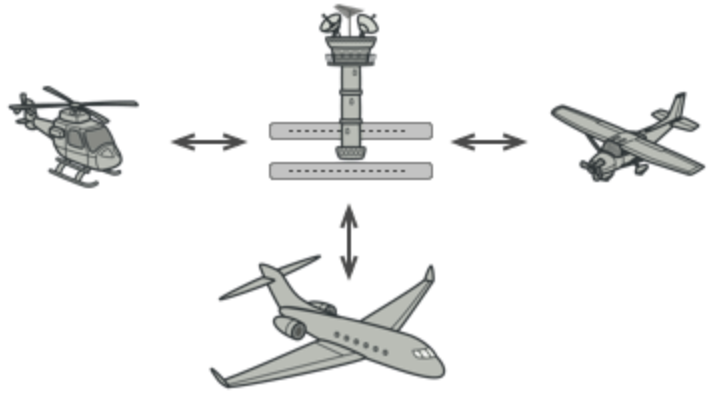
Mediator (Controller)

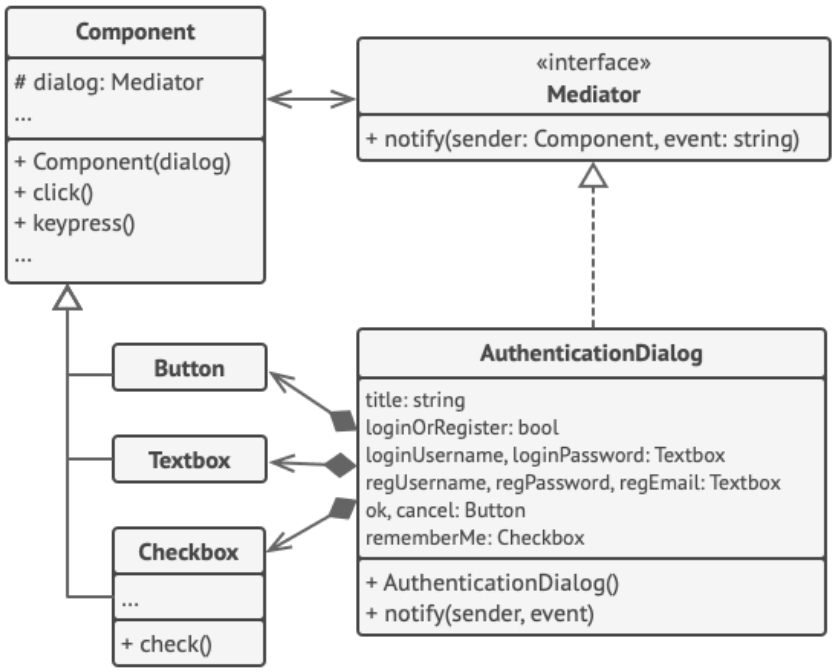
- Restricts direct communications between objects and forces them to collaborate only via a mediator - reduces chaotic dependencies
- Use Mediator when making a change is hard due to tight coupling
- Can be implemented using Observer-ish pattern but also using different methods
- Problem:



- Solution - cease all direct communications:







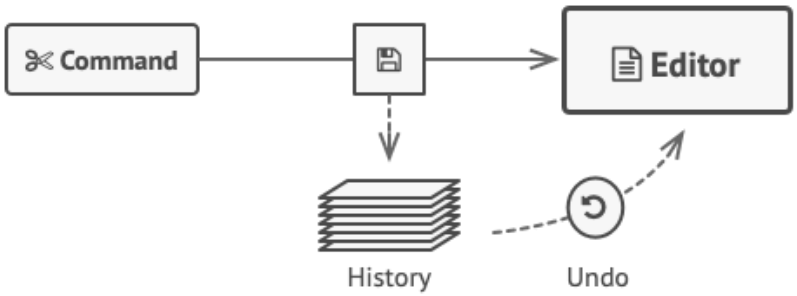
- Pros:
    - Single Responsibility Principle

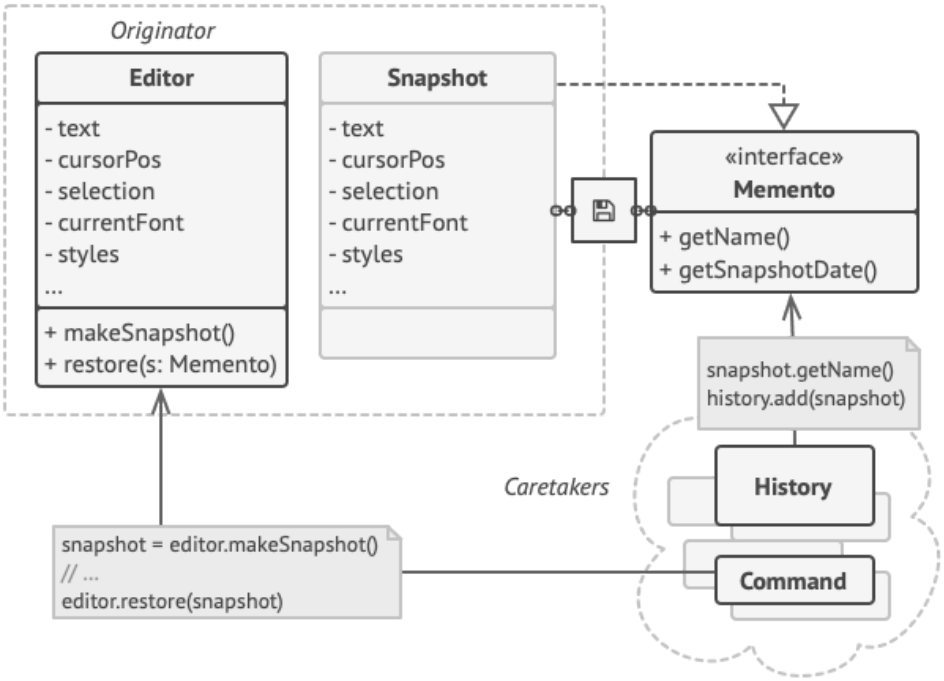
    - Open-Closed Principle - new mediators
    - Reduces coupling
    - Reusability of components is more efficient
- Cons:
    - Mediator may evolve into God Object

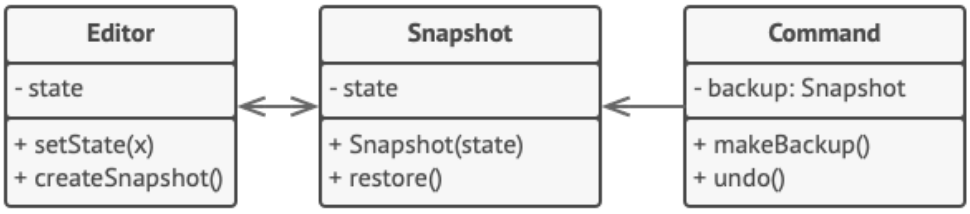
```java
// Mediator interface
interface ChatMediator {
    void sendMessage(User sender, String message);
}

// Concrete Mediator
class ConcreteChatMediator implements ChatMediator {
    private List<User> users;

    public ConcreteChatMediator() {
        this.users = new ArrayList<>();
    }

    public void addUser(User user) {
        users.add(user);
    }

    @Override
    public void sendMessage(User sender, String message) {
        for (User user : users) {
            // Send the message to all users except the sender
            if (user != sender) {
                user.receiveMessage(message);
            }
        }
    }
}

// Colleague interface
interface User {
    void sendMessage(String message);
    void receiveMessage(String message);
}
```

```java
// Concrete Colleague
class ConcreteUser implements User {
    private String name;
    private ChatMediator mediator;

    public ConcreteUser(String name, ChatMediator mediator) {
        this.name = name;
        this.mediator = mediator;
        mediator.addUser(this);
    }

    @Override
    public void sendMessage(String message) {
        System.out.println(name + " sends message: " + message);
        mediator.sendMessage(this, message);
    }

    @Override
    public void receiveMessage(String message) {
        System.out.println(name + " receives message: " + message);
    }
}

// Client code
public class MediatorExample {
    public static void main(String[] args) {
        // Create a chat mediator
        ChatMediator chatMediator = new ConcreteChatMediator();

        // Create users and associate them with the mediator
        User user1 = new ConcreteUser("User 1", chatMediator);
        User user2 = new ConcreteUser("User 2", chatMediator);
        User user3 = new ConcreteUser("User 3", chatMediator);

        // Users send and receive messages through the mediator
        user1.sendMessage("Hello, everyone!");
        user2.sendMessage("Hi there!");
        user3.sendMessage("Nice to meet you!");
    }
}
```

Memento

- Save and restore the previous state of an object
- Problem: Saving content of WYSIWYG editor and then undo
- 

- 

- 

- Pros:
    - We can produce snapshots of object's state without violating its encapsulation

    - We can introduce a caretaker to maintain the history of states - simplify originator's code
- Cons:
    - More memory resource heavy, if used too often
    - Caretakers should track originator's lifecycle to destroy obsolete mementos
    - Most dynamic languages can't guarantee that the state within memento is untouched (PHP, JS, Python, ...)

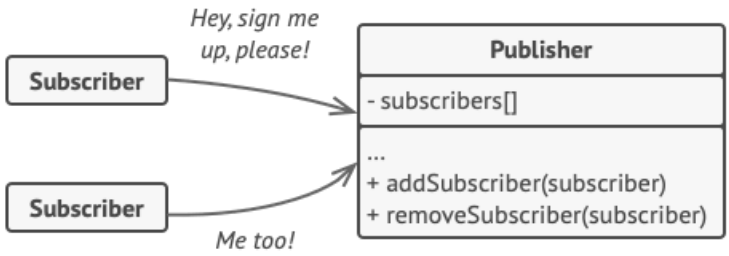
```
// Originator
class TextEditor {
    private StringBuilder text;

    public TextEditor() {
        this.text = new StringBuilder();
    }

    public void addText(String addedText) {
        text.append(addedText);
    }

    public String getText() {
        return text.toString();
    }

    public TextMemento save() {
        return new TextMemento(text.toString());
    }

    public void restore(TextMemento memento) {
        this.text = new StringBuilder(memento.getState());
    }
}

// Memento
class TextMemento {
    private String state;

    public TextMemento(String state) {
        this.state = state;
    }
```

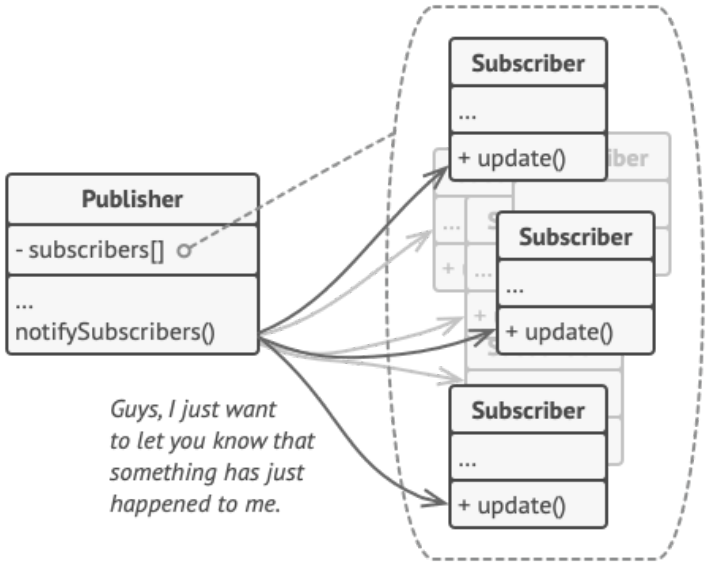
```
    public String getState() {
        return state;
    }
}

// Caretaker
class TextEditorHistory {
    private List<TextMemento> history = new ArrayList<>();

    public void saveState(TextEditor textEditor) {
        history.add(textEditor.save());
    }

    public void undo(TextEditor textEditor) {
        if (!history.isEmpty()) {
            TextMemento lastMemento = history.remove(history.size() - 1);
            textEditor.restore(lastMemento);
        }
    }
}

// Client code
public class MementoExample {
    public static void main(String[] args) {
        // Create a text editor
        TextEditor textEditor = new TextEditor();
        TextEditorHistory history = new TextEditorHistory();

        // Add text and save states
        textEditor.addText("Hello, ");
        history.saveState(textEditor);

        textEditor.addText("Memento ");
        history.saveState(textEditor);

        textEditor.addText("Pattern!");
        history.saveState(textEditor);

        // Print current text
        System.out.println("Current Text: " + textEditor.getText());

        // Undo changes
```

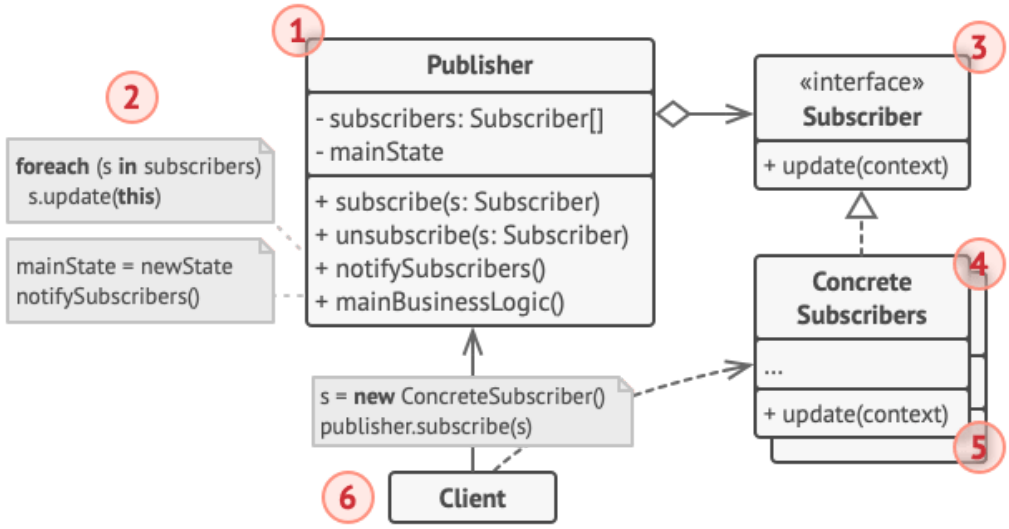
```
        history.undo(textEditor);
        System.out.println("After Undo: " + textEditor.getText());

        history.undo(textEditor);
        System.out.println("After Undo: " + textEditor.getText());

        // Redo changes
        history.saveState(textEditor); // Save the current state before
redo
        history.undo(textEditor);
        System.out.println("After Redo: " + textEditor.getText());
    }
}
```

Observer (Event-Subscriber, Listener)

- Subscription mechanism to notify multiple objects about events they are observing
- Problem: A Customer wants to know when the iPhone will be available in the store. The store doesn't want to send spam to inform about the availability.
- Use Observer when changes to the state may require changing other objects

- 

- 

- 

- **Receiving the state of the changed state**
  - Subscribers will only be notified about the change happening, not what has changed
  - Pull model - each subscriber will pull info from the publisher explicitly

    - Push model - publisher sends the change alongside the notification
    - Aspect of interest model - during subscription, each subscriber defines an aspect of interest and notifications are called with an aspect sign
- Pros:
    - Open-Closed Principle - new subscribers
    - We can establish new relations between objects at runtime
- Cons:
    - Subscribers are notified in random order

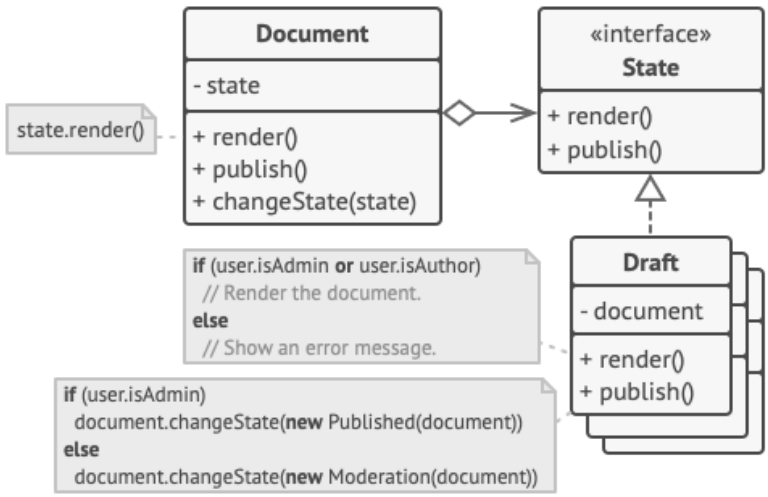
```
// Subject (Observable)
interface WeatherStation {
    void addObserver(WeatherDisplay observer);
    void removeObserver(WeatherDisplay observer);
    void notifyObservers();
}

// Concrete Subject
class ConcreteWeatherStation implements WeatherStation {
    private int temperature;
    private List<WeatherDisplay> observers;

    public ConcreteWeatherStation() {
        this.observers = new ArrayList<>();
    }

    public void setTemperature(int temperature) {
        this.temperature = temperature;
        notifyObservers();
    }

    @Override
    public void addObserver(WeatherDisplay observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(WeatherDisplay observer) {
        observers.remove(observer);
    }

    @Override
```

```
    public void notifyObservers() {
        for (WeatherDisplay observer : observers) {
            observer.update(temperature);
        }
    }
}

// Observer
interface WeatherDisplay {
    void update(int temperature);
}

// Concrete Observer 1
class CurrentConditionsDisplay implements WeatherDisplay {
    @Override
    public void update(int temperature) {
        System.out.println("Current Conditions Display: Temperature is " +
temperature + " degrees Celsius");
    }
}

// Concrete Observer 2
class ForecastDisplay implements WeatherDisplay {
    @Override
    public void update(int temperature) {
        System.out.println("Forecast Display: Expect warmer weather with
temperature " + (temperature + 5) + " degrees Celsius");
    }
}

// Client code
public class ObserverExample {
    public static void main(String[] args) {
        // Create a weather station
        ConcreteWeatherStation weatherStation = new
ConcreteWeatherStation();

        // Create weather displays (observers)
        WeatherDisplay currentConditionsDisplay = new
CurrentConditionsDisplay();
        WeatherDisplay forecastDisplay = new ForecastDisplay();
```

```
        // Register observers with the weather station
        weatherStation.addObserver(currentConditionsDisplay);
        weatherStation.addObserver(forecastDisplay);

        // Simulate a change in temperature
        weatherStation.setTemperature(25);
    }
}
```

State

- Altering an object's behavior based on its state without changing its class
- Finite State Machine
    - a finite number of states a program can be in
    - switching between states based on switching rules (transitions)
    - behaves differently in each state
- 



-
- Pros:
    - Single Responsibility Principle
    - Open-Closed Principle - adding new states
    - Simplifying the code by eliminating state machine conditionals
- Cons:
    - May be an overkill if we only need a state machine with few states that rarely changes

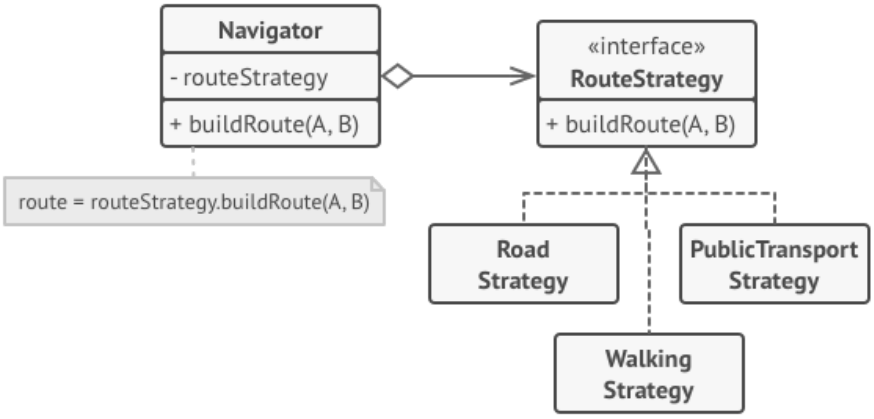
```
// Context class that maintains a reference to the current state
class Context {
    private State state;

    public Context() {
        // Set the initial state
        this.state = new ConcreteStateA();
    }

    public void setState(State state) {
        this.state = state;
    }

    public void request() {
        state.handle();
    }
}

// State interface
```

```java
interface State {
    void handle();
}

// ConcreteStateA class implementing the State interface
class ConcreteStateA implements State {
    @Override
    public void handle() {
        System.out.println("Handling request in State A");
    }
}

// ConcreteStateB class implementing the State interface
class ConcreteStateB implements State {
    @Override
    public void handle() {
        System.out.println("Handling request in State B");
    }
}

public class StatePatternExample {
    public static void main(String[] args) {
        // Create a context object
        Context context = new Context();

        // Perform requests with different states
        context.request(); // Output: Handling request in State A

        // Change the state
        context.setState(new ConcreteStateB());

        // Perform requests with the new state
        context.request(); // Output: Handling request in State B
    }
}
```

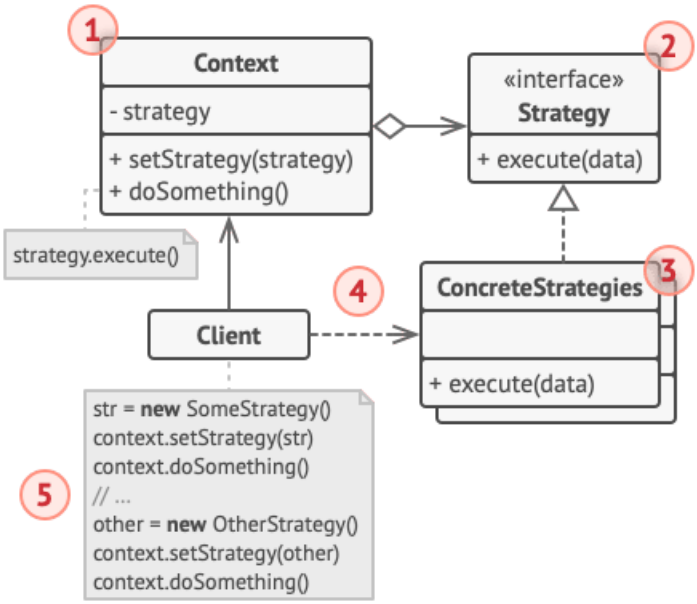
Strategy

- Define a family of algorithms, put them in different classes and make their objects interchangeable

- Problem: People want to travel your map by car, bike, walking, public transports, … Gotta build different route for each of them



- A class that does something specific in a lot of different ways -> extract those algorithms into strategies -> original class (context) contains references to the strategies used



- Pros:
  - We can swap algorithms at runtime
  - We can isolate implementation of an algorithm
  - We can replace inheritance with composition
  - Open-Closed Principle - adding new strategies
- Cons:

- Overkill when we have only few algorithms that rarely change
- Clients must be aware of the differences between strategies to select a proper one
- Anonymous functions do the same without boiler code

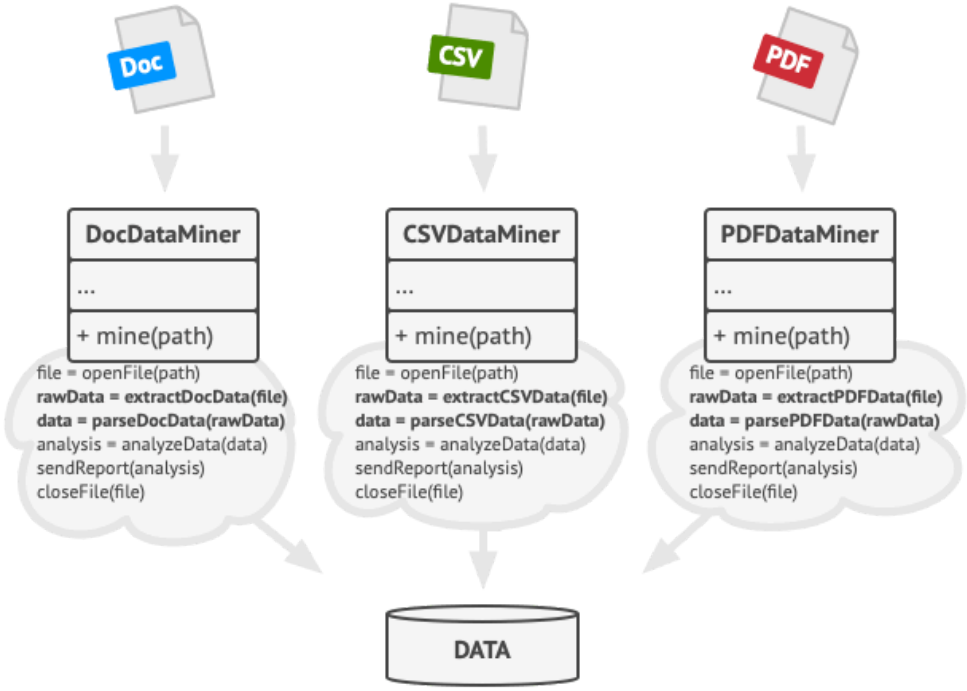
```
// Strategy interface
interface AttackStrategy {
    void attack();
}

// Concrete Strategy 1: SwordAttack
class SwordAttack implements AttackStrategy {
    @Override
    public void attack() {
        System.out.println("Attacking with a sword!");
        // Additional sword attack logic
    }
}

// Concrete Strategy 2: MagicAttack
class MagicAttack implements AttackStrategy {
    @Override
    public void attack() {
        System.out.println("Casting a magic spell!");
        // Additional magic attack logic
    }
}

// Context
class Character {
    private String name;
    private AttackStrategy attackStrategy;

    public Character(String name) {
        this.name = name;
    }

    public void setAttackStrategy(AttackStrategy attackStrategy) {
        this.attackStrategy = attackStrategy;
    }

    public void performAttack() {
```

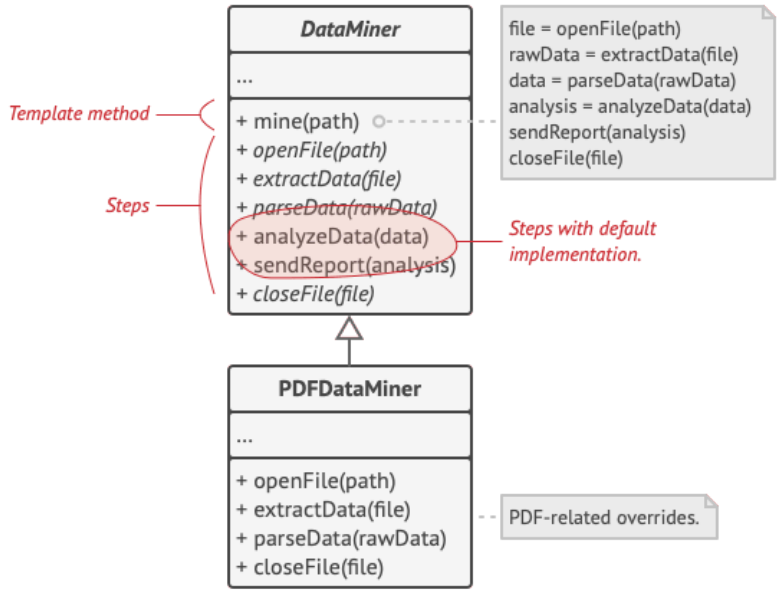
```
        System.out.println(name + " performs attack:");
        if (attackStrategy != null) {
            attackStrategy.attack();
        } else {
            System.out.println("No attack strategy set.");
        }
    }
}

// Client code
public class RPGExample {
    public static void main(String[] args) {
        // Create RPG characters
        Character knight = new Character("Knight");
        Character mage = new Character("Mage");

        // Set attack strategies dynamically
        knight.setAttackStrategy(new SwordAttack());
        mage.setAttackStrategy(new MagicAttack());

        // Perform attacks
        knight.performAttack();  // Knight attacks with a sword
        mage.performAttack();    // Mage attacks with magic

        // Change attack strategies dynamically
        knight.setAttackStrategy(new MagicAttack());
        mage.setAttackStrategy(new SwordAttack());

        // Perform new attacks
        knight.performAttack();  // Knight attacks with magic
        mage.performAttack();    // Mage attacks with a sword
    }
}
```

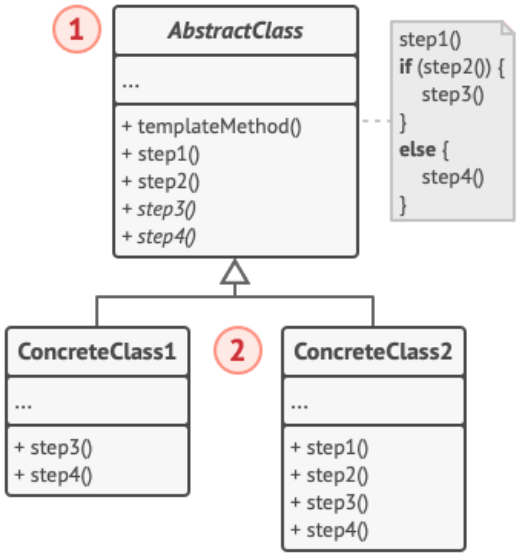
Template Method

- The skeleton of an algorithm in superclass with subclasses overriding some of its steps without changing the whole structure

- Use Template Method when you want to let clients extend only particular steps of an algorithm
- Problem: DataMiner that is fed multiple formats (CSV, Doc, PDF, ...)





- 

- Pros:
  - Clients can override only certain parts of a large algorithm
  - Duplicate code can be pulled into superclass
- Cons:
  - Clients may be limited by the provided skeleton
  - We might violate Liskov-Substitution Principle - suppressing default steps via subclass
  - Template methods are harder to maintain the more steps they have

```
// Abstract class defining the template method
```

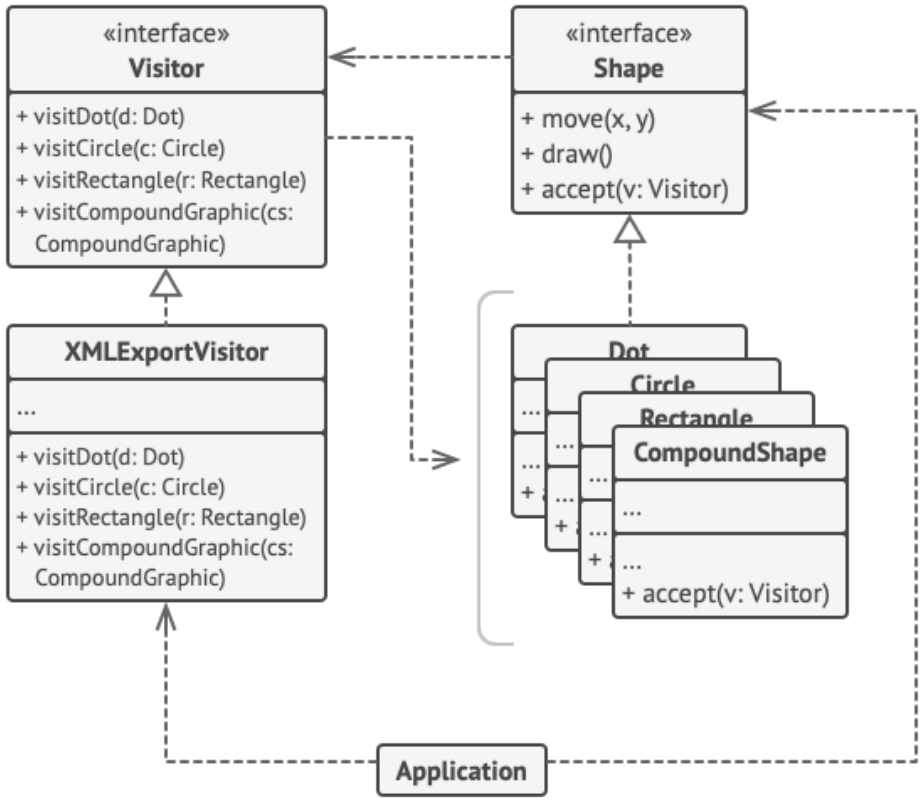
```
abstract class HouseBuilder {
    // Template method that defines the steps to build a house
    public final void buildHouse() {
        constructFoundation();
        constructWalls();
        addWindows();
        addDoors();
        paintHouse();
        decorateHouse();
        addUtilities();
        System.out.println("House construction completed.");
    }

    // Abstract methods to be implemented by subclasses
    protected abstract void constructFoundation();
    protected abstract void constructWalls();
    protected abstract void addWindows();
    protected abstract void addDoors();

    // Hook methods with default implementation (optional to override)
    protected void paintHouse() {
        System.out.println("Default paint color applied.");
    }

    protected void decorateHouse() {
        System.out.println("Basic decoration added.");
    }

    // Hook method to be optionally overridden by subclasses
    protected void addUtilities() {
        // Default: no additional utilities
    }
}

// Concrete class 1: WoodenHouse
class WoodenHouse extends HouseBuilder {
    @Override
    protected void constructFoundation() {
        System.out.println("Constructing wooden foundation.");
    }

    @Override
```

```
    protected void constructWalls() {
        System.out.println("Constructing wooden walls.");
    }

    @Override
    protected void addWindows() {
        System.out.println("Adding glass windows to the wooden house.");
    }

    @Override
    protected void addDoors() {
        System.out.println("Adding wooden doors to the wooden house.");
    }

    @Override
    protected void decorateHouse() {
        System.out.println("Adding rustic wooden decoration.");
    }
}

// Concrete class 2: ConcreteHouse
class ConcreteHouse extends HouseBuilder {
    @Override
    protected void constructFoundation() {
        System.out.println("Constructing concrete foundation.");
    }

    @Override
    protected void constructWalls() {
        System.out.println("Constructing concrete walls.");
    }

    @Override
    protected void addWindows() {
        System.out.println("Adding metal-framed windows to the concrete
house.");
    }

    @Override
    protected void addDoors() {
        System.out.println("Adding steel doors to the concrete house.");
    }
```

```
    @Override
    protected void paintHouse() {
        System.out.println("Applying beige paint color to the concrete
house.");
    }

    @Override
    protected void addUtilities() {
        System.out.println("Adding plumbing and electrical utilities to the
concrete house.");
    }
}

// Client code
public class TemplateMethodExample {
    public static void main(String[] args) {
        System.out.println("Building a Wooden House:");
        HouseBuilder woodenHouseBuilder = new WoodenHouse();
        woodenHouseBuilder.buildHouse();

        System.out.println("\nBuilding a Concrete House:");
        HouseBuilder concreteHouseBuilder = new ConcreteHouse();
        concreteHouseBuilder.buildHouse();
    }
}
```

Visitor

- Separate algorithms from the objects on which they operate
- Problem: We want to perform render() on all kinds of objects but we don't want to modify every object
- Use Visitor when you need to perform an operation on all elements of a complex object structure
- We create a Visitor that performs the action (render) upon receiving the object
- The action (render) is called by calling object.accept(visitor), so we can still use polymorphism
    - visitor.accept(this);



-
- Pros:
    - Open-Closed Principle - adding new behavior to the visitor
    - Single Responsibility Principle
    - Visitor can accumulate useful information while working with various objects
- Cons:
    - Need to update all visitors each time a class gets added/removed to the element hierarchy
    - Visitors may lack necessary access to private fields, methods etc.

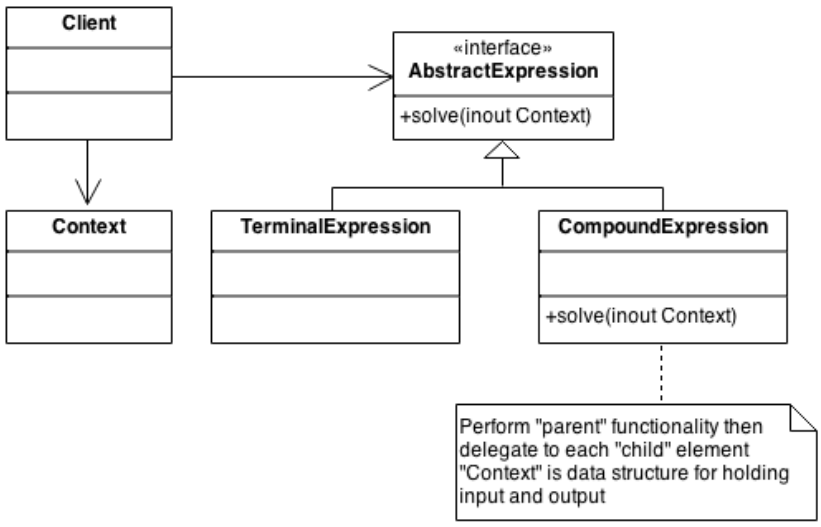
```
// Element interface
interface Renderable {
    void accept(RenderVisitor visitor);
}

// Concrete Element 1: Character
class Character implements Renderable {
    private String name;

    public Character(String name) {
```

```
        this.name = name;
    }

    @Override
    public void accept(RenderVisitor visitor) {
        visitor.visitCharacter(this);
    }

    public String getName() {
        return name;
    }
}

// Concrete Element 2: Obstacle
class Obstacle implements Renderable {
    private String type;

    public Obstacle(String type) {
        this.type = type;
    }

    @Override
    public void accept(RenderVisitor visitor) {
        visitor.visitObstacle(this);
    }

    public String getType() {
        return type;
    }
}

// Visitor interface
interface RenderVisitor {
    void visitCharacter(Character character);
    void visitObstacle(Obstacle obstacle);
}

// Concrete Visitor 1: SimpleRenderVisitor
class SimpleRenderVisitor implements RenderVisitor {
    @Override
    public void visitCharacter(Character character) {
        System.out.println("Rendering Character: " + character.getName());
```

```
    }

    @Override
    public void visitObstacle(Obstacle obstacle) {
        System.out.println("Rendering Obstacle: " + obstacle.getType());
    }
}

// Concrete Visitor 2: FancyRenderVisitor
class FancyRenderVisitor implements RenderVisitor {
    @Override
    public void visitCharacter(Character character) {
        System.out.println("Fancy Rendering Character: " +
character.getName());
        // Additional fancy rendering logic for characters
    }

    @Override
    public void visitObstacle(Obstacle obstacle) {
        System.out.println("Fancy Rendering Obstacle: " +
obstacle.getType());
        // Additional fancy rendering logic for obstacles
    }
}
```

Interpreter

- A way to define grammar or include language elements in program, such as expression interpreters
- Use interpreter when dealing with languages and grammars that can be expressed in a hierarchical way



-
- Example
    - expression = Addition(Number(5), Subtraction(Number(3), Number(2)));
      result = expression.interpret();
- Pros:
    - Great pattern for relatively simple grammar interpretation that doesn't evolve much
- Cons:
    - With complex grammar, the pattern becomes harder to maintain

```
interface Expression
{
    boolean interpreter(String con);
}

class TerminalExpression implements Expression
{
    String data;

    public TerminalExpression(String data)
    {
        this.data = data;
    }

    public boolean interpreter(String con)
    {
        if(con.contains(data)) {
```

```
            return true;
        } else {
            return false;
        }
    }
}

class OrExpression implements Expression
{
    Expression expr1;
    Expression expr2;

    public OrExpression(Expression expr1, Expression expr2)
    {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public boolean interpreter(String con)
    {
        return expr1.interpreter(con) || expr2.interpreter(con);
    }
}

class AndExpression implements Expression
{
    Expression expr1;
    Expression expr2;

    public AndExpression(Expression expr1, Expression expr2)
    {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public boolean interpreter(String con)
    {
        return expr1.interpreter(con) && expr2.interpreter(con);
    }
}

// Driver class
class InterpreterPattern
{
```

```
    public static void main(String[] args)
    {
        Expression person1 = new TerminalExpression("Kushagra");
        Expression person2 = new TerminalExpression("Lokesh");
        Expression isSingle = new OrExpression(person1, person2);

        Expression vikram = new TerminalExpression("Vikram");
        Expression committed = new TerminalExpression("Committed");
        Expression isCommitted = new AndExpression(vikram, committed);

        System.out.println(isSingle.interpreter("Kushagra"));
        System.out.println(isSingle.interpreter("Lokesh"));
        System.out.println(isSingle.interpreter("Achint"));

        System.out.println(isCommitted.interpreter("Committed, Vikram"));
        System.out.println(isCommitted.interpreter("Single, Vikram"));

    }
}
```

Non-GoF design patterns

Other design patterns not listed in the Gang-Of-Four book of design patterns.

Dependency injection

- Technique that makes a class independent of its dependencies
- By decoupling the usage of an object from its creation
- Inject in constructor on in parameter or with setter
- It helps to follow Dependency Inversion Principle and Single Responsibility Principle
- It helps to replace dependencies without changing the class
- DI is implemented in many modern frameworks (DI container performs the injections)
- A class does not ask for the dependency, a class is provided (supplied) with the dependency it needs
- First we create an instance of Engine and then use it to construct an instance of Car

    - A car is then reusable, we can pass different Engines to the car
    - We can easily test the car, pass mock engines etc
- Many dependencies may make the manual dependency injection tedious
    - Automated DI - a library or FW that does this for us
- Pros:
    - reusability of classes, decoupling of dependencies
    - easier to swap out implementations
    - Dependency Inversion, SRP
    - we work with configuration, not dependencies
    - ease of refactoring
    - ease of testing

Double checked locking

- Performance boost by checking the locking condition beforehand - reduces the number of lock acquisitions
- If there is only one check for creating a singleton, another thread can create new instance during the first instance creation
- Check - possibility of multiple thread creating it at the same time
- Check-Check - same problem
- Check-Lock - we could lock after it was created, then overwrite it
- Lock-Check - works, but locks are expensive
- Check-Lock-Check - correct
- Pros:
    - Secure
    - Fast

Lazy loading

- delayed load or initialization of resource until they are needed
- implementation
    - lazy initialization - objects are set to null, data are loaded after they are invoked

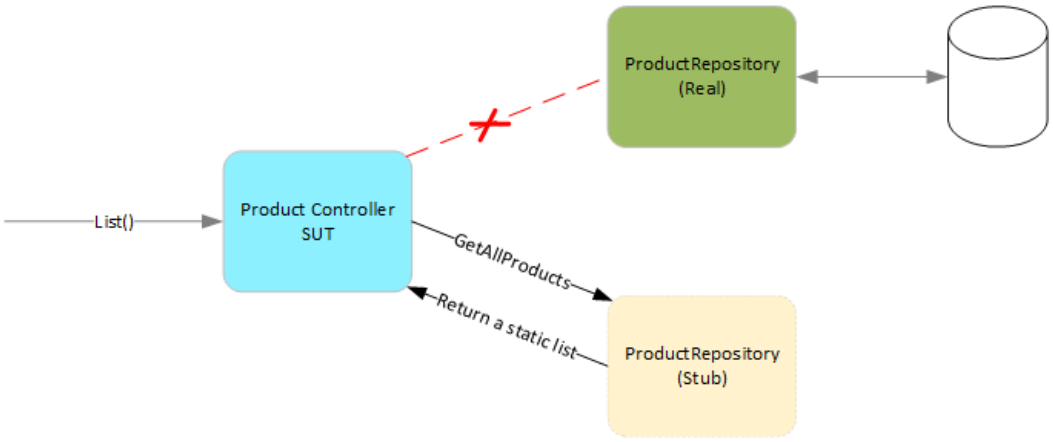
    - virtual proxy - we access virtual object with the same interface before accessing the actual object
    - ghost - load the object with partial state, load data when property is is called
    - value holder - a generic object that handles lazy loading and appears in place of objects data fields
- Pros:
    - reduces load time
    - bandwidth conservation
    - improve performance
    - save system resources
- Cons:
    - may be a bit slow, adds overhead

Marker interface

- An empty interface, no need to add implementation, used to indicate a special treatment of an object
- For example: Serializable, Cloneable (this object is available for this operation)

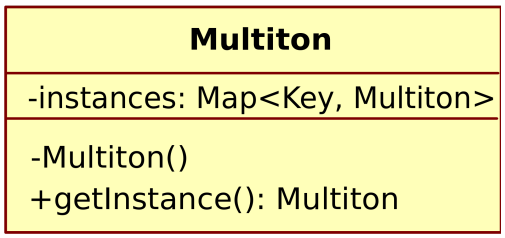
Mock object

- An object that substitutes for a real object

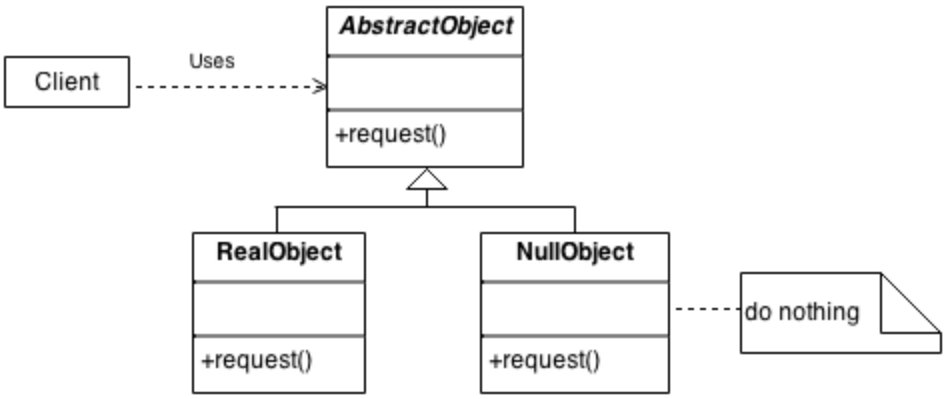


- Great enablers of testing and good design:
    - Dependencies are reduced
    - Tests run faster (no need to hit the actual DB)

Multiton

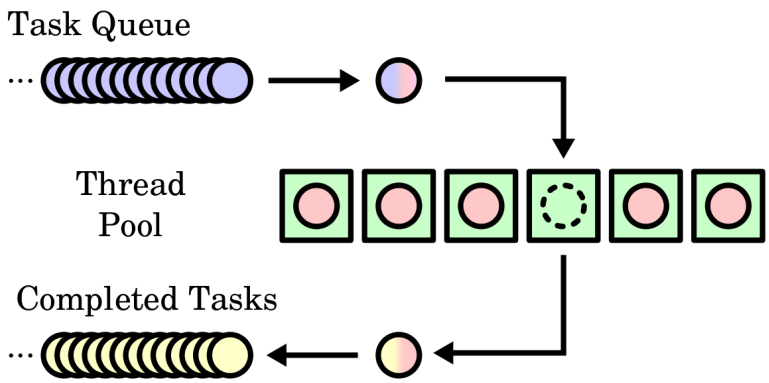
- Ensures that there are predefined amount of instances available globally
- Generalization of Singleton
- 

Null object

- A substitution for objects when they are undefined
- Implements a known interface but does nothing
- Usually implemented as Singleton
- 

Thread pool

- N threads, M jobs, running concurrently
- Avoiding overhead associated with thread management

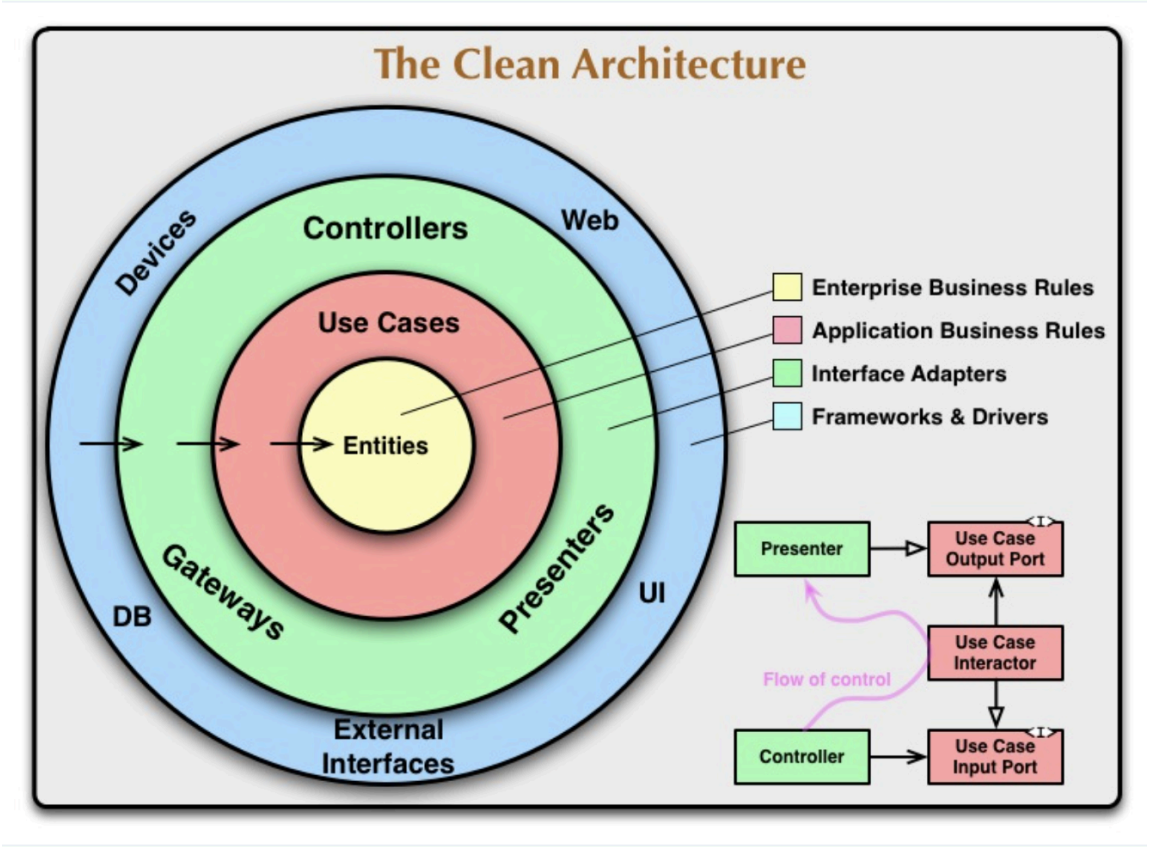


Architectures

Clean architecture

- Independent of Frameworks - does not depend on libraries, ...
- Testable - we can mock UI, DB, external components, ...
- Independent of UI - UI should be easily changeable
- Independent of Database - DB should be changeable

- Independent of any external agency - our business rules shouldn't know anything about the outside world



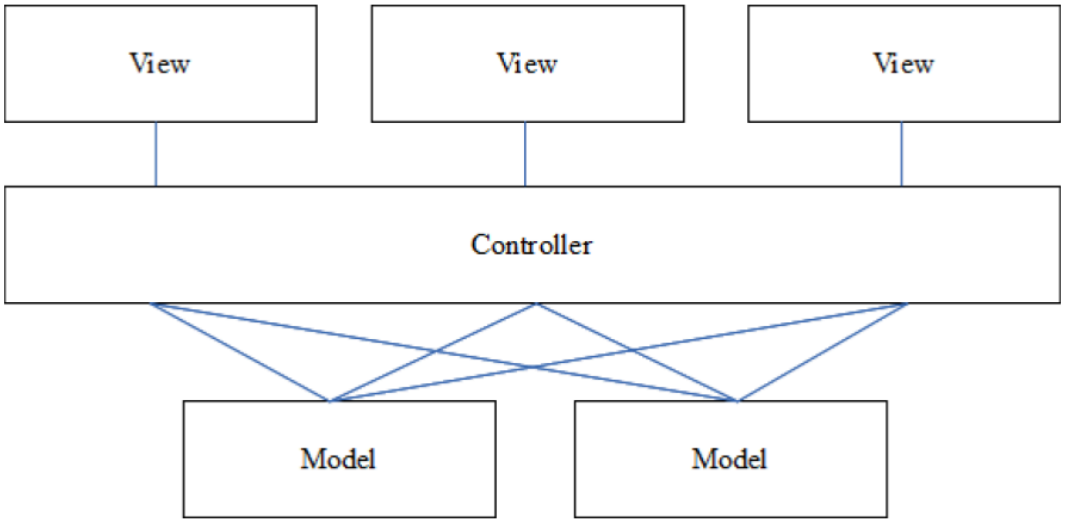
  - **Dependency rule** - source code dependencies can only point inwards, inner circles don't know anything about the outer circles (data formats in outer world shouldn't be used in the inner policies), inner circle won't be affected by the change of outer circle
  - **Entities** - encapsulate enterprise business rules, business objects, least likely to change when something in outer circle changes
  - **Use Cases** - software here contains application specific business rules, implements use cases of the system, orchestrate flow of data to and from entities
  - **Interface adapters** - set of adapters to convert data for convenient use for use cases and entities
  - **Frameworks and drivers** - Web frameworks, DBs, libraries

- **Crossing boundaries** - if we need use case to call presenter, we make it call an interface in the inner circle and make a presenter implement this interface

Architectural patterns

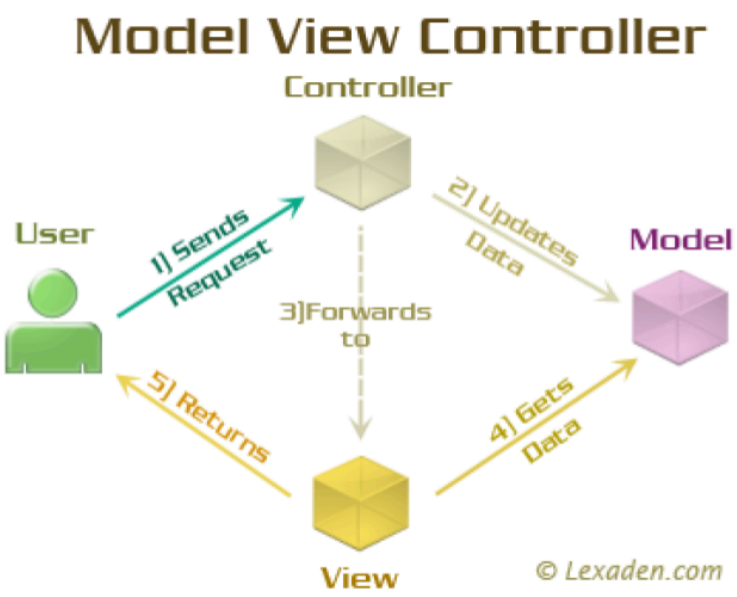
MVC

- Model View Controller
- A design pattern as well as architecture



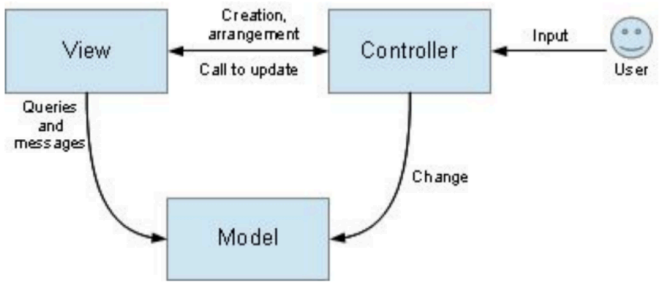
13

- Usually bent to one's needs
- What stays the same: Separation of concerns, loose coupling, high cohesion, ...



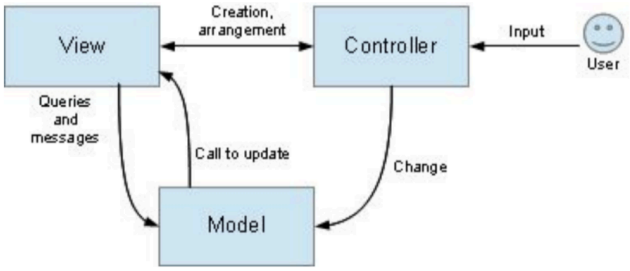
24

-
- **Model** - handles business logic, verifies data
- **Controller** - handling user input, update Model
- **View** - GUI, user interface
- **Implementations**
    - Controller updates View when changes are made



14

- Model tells View to update when change has been made (usually uses Observer pattern)

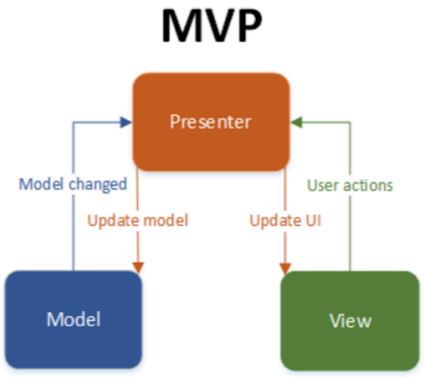


15

- Pros
  - application is structured, readability, maintainability
  - business logic separated from view logic
  - multiple views for one model etc.
- Cons
  - tight coupling of View and Controller, harder testing

MVP

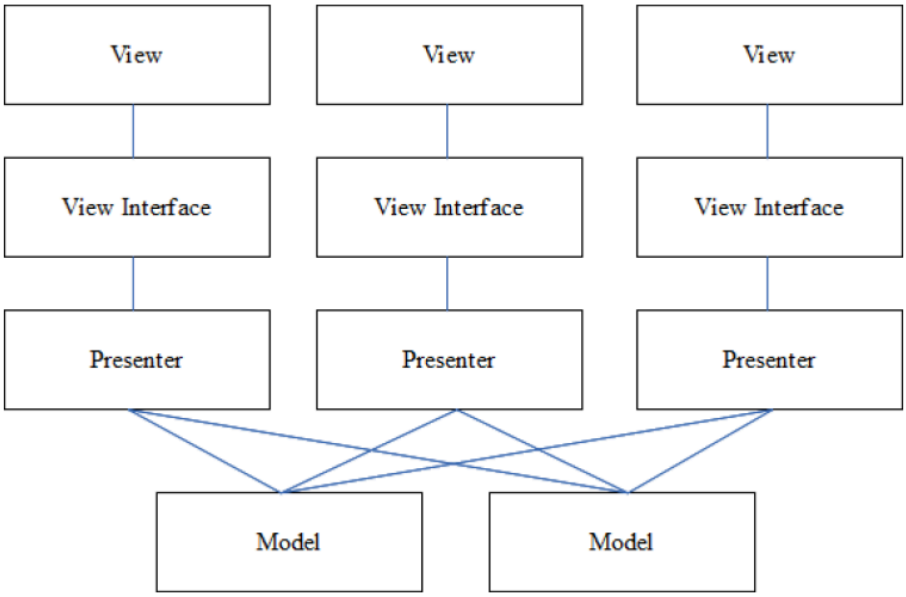
- Model View Presenter
- Evolution of MVC, improves SoC and facilitate automatic unit testing



18

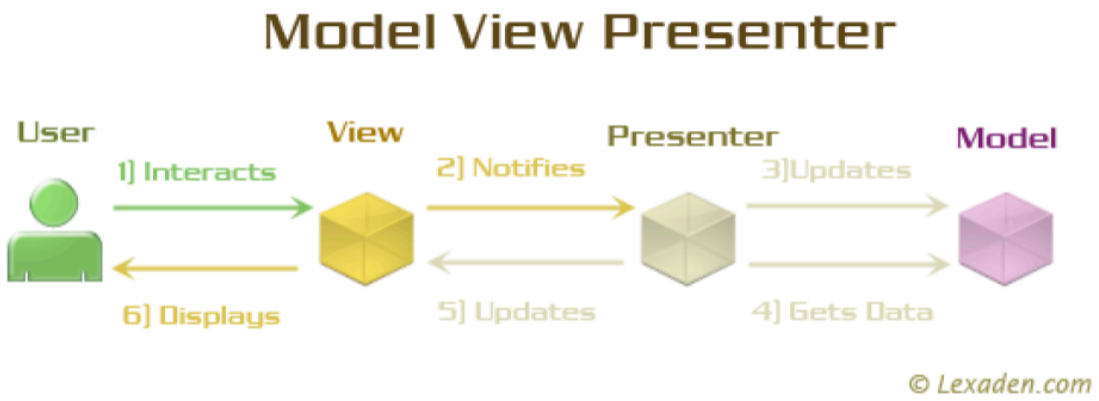
- **Model** - data layer, handling business logic, DB, APIs, caching, ...

- **Presenter** - middleman, retrieves data from Model, returns it formatted to View, processes View interaction, works closely with the View, has a corresponding View and their interaction is defined by an interface
- **View** - only calls Presenter methods on interaction, presents the data in a way decided by Presenter



17

-
- Pros
    - Clear separation of responsibilities, readability, maintainability
    - Modularity, switching between View implementations, ...
    - Easier testing, mocking other components

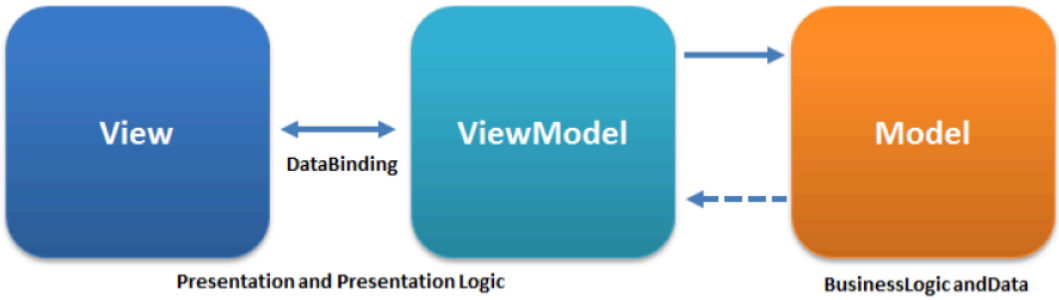


24

-

MVVM

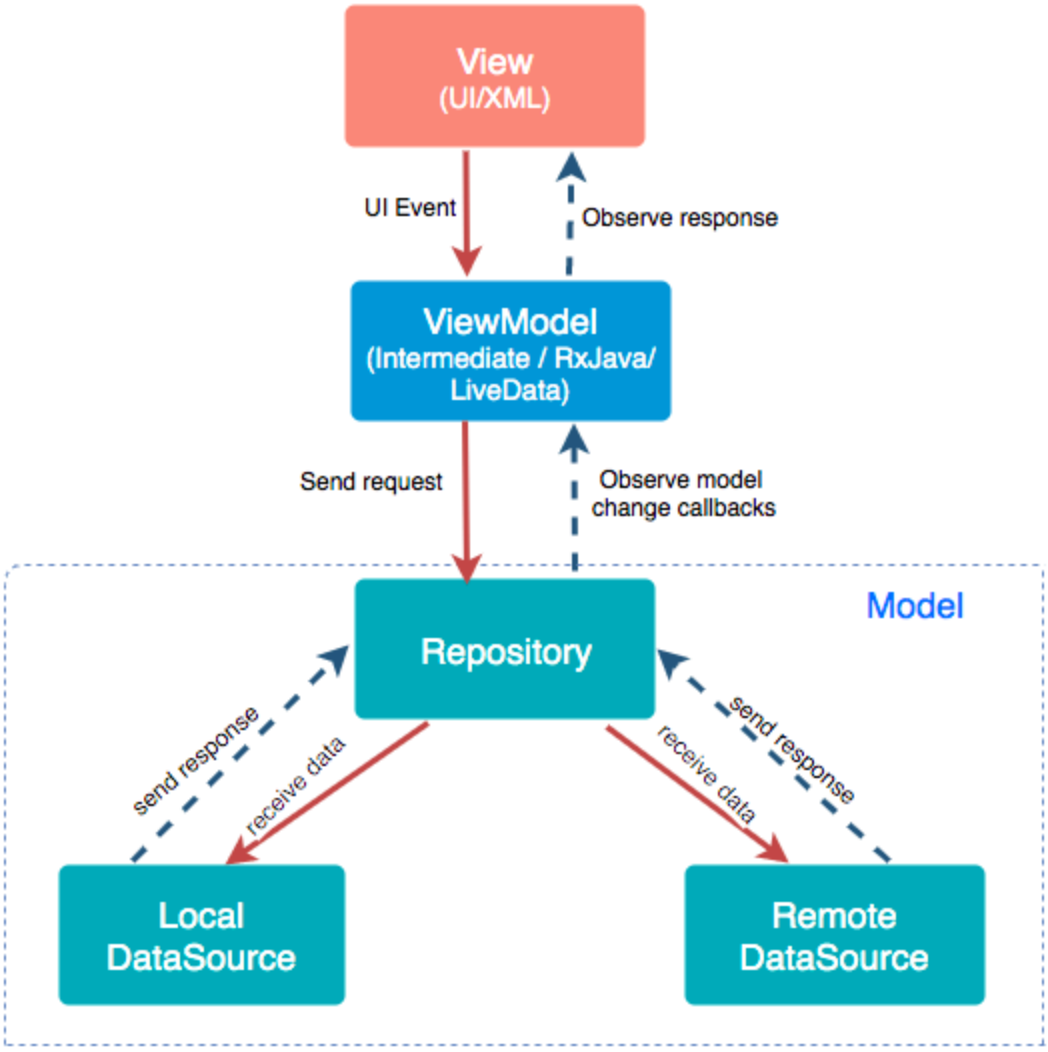
- Model View ViewModel



20

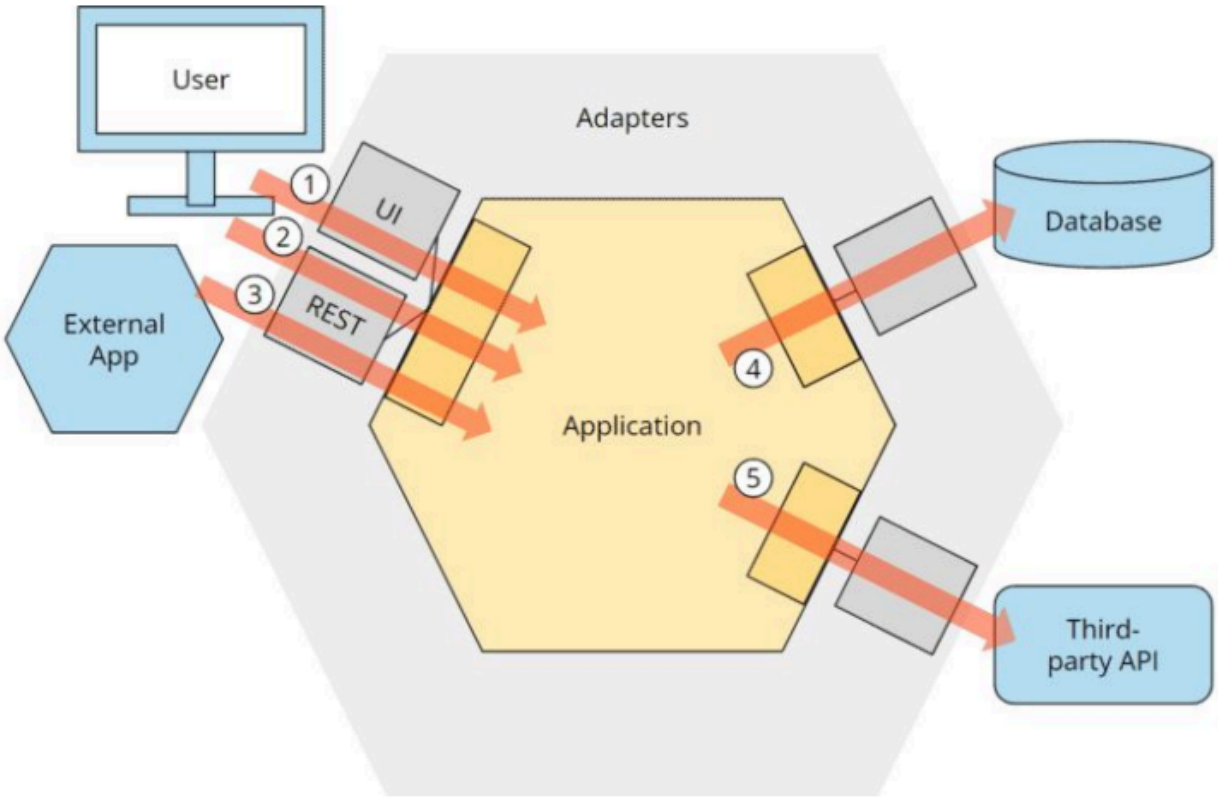
-
- Similar to MVP
- Separates View from Model and communicate via ViewModel
- **ViewModel**
    - doesn't know anything about View

    - only provides data, View is subscribed so it can rerender itself on change
- Implementation usually uses Reactive programming, Data binding



Architectures

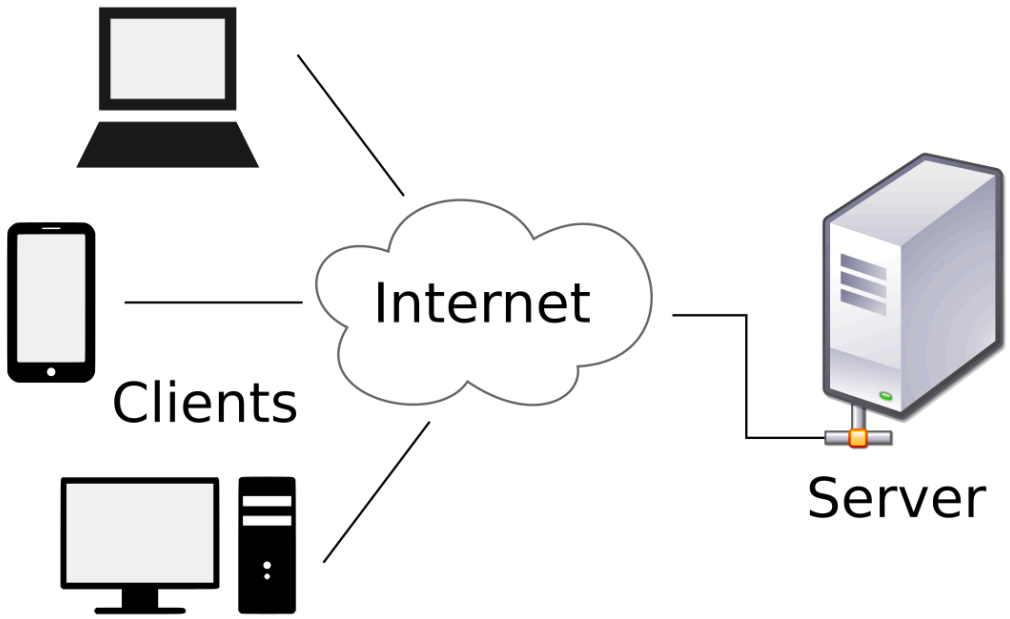
Hexagonal architecture



Client-Server architecture

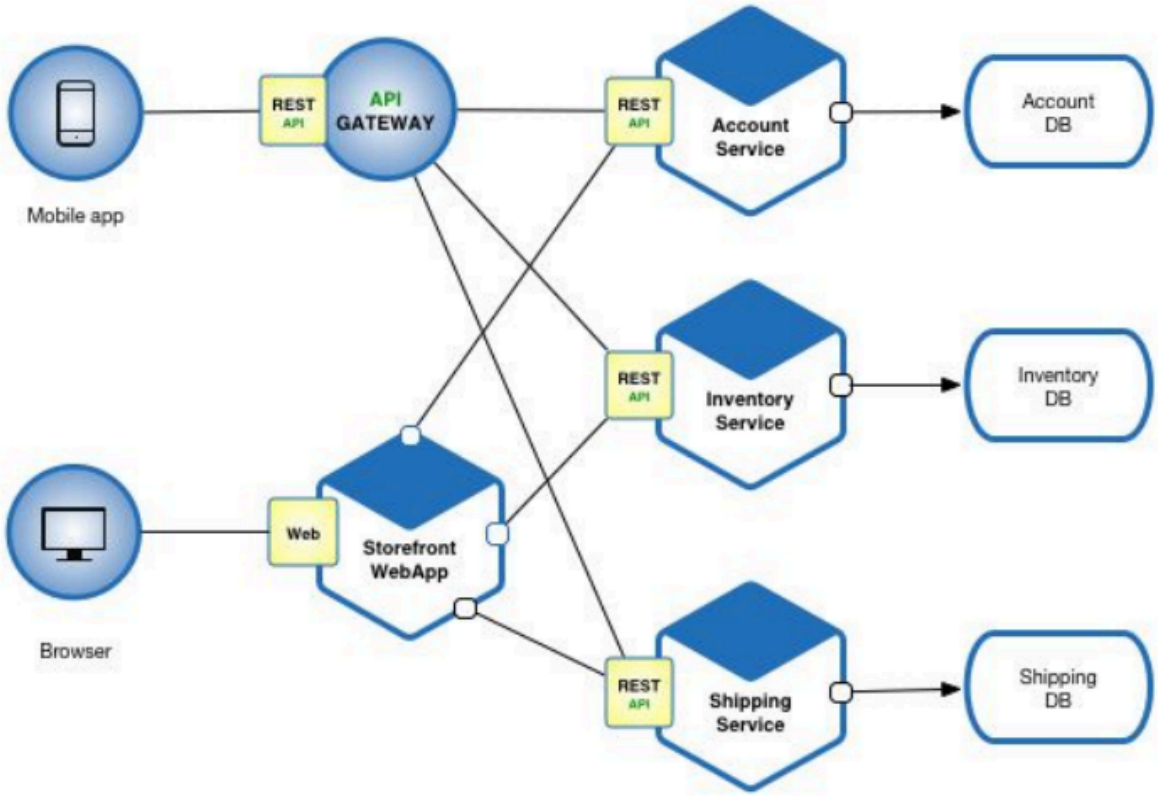
- Client and server usually connected over network
- Clients initiates requests to the server
- Server handles request, performs action and returns response

- Multiple clients for one server



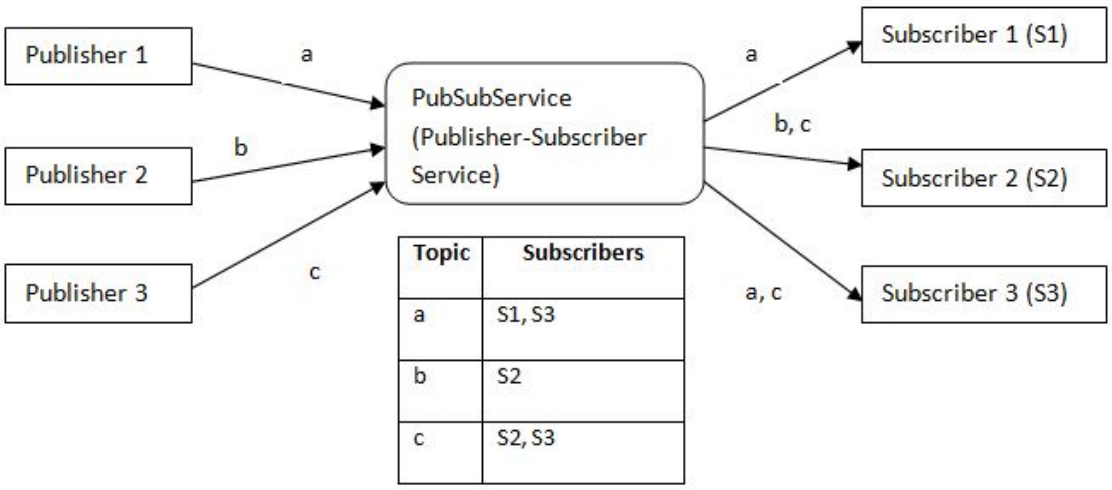
- Pros
    - SoC, clients responsible for user interface, server responsible for logic and storage
    - Scalability, server can handle requests concurrently or distribute the load across multiple servers
- Cons
    - Requires network connection, latency, other network issues
    - Server is a single point of failure
- Use Cases
    - Web application
    - Desktop applications
    - Mobile apps
    - Online games

Microservice architecture



Publish-Subscribe architecture

- Publishers categorize published messages, subscribers receive them
- Subscribers usually receive only a subset of the published messages



Topic	Subscribers
a	S1, S3
b	S2
c	S2, S3

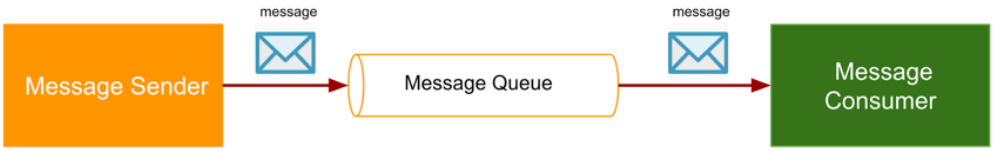
- **Bus/Broker topology**
    - publishers post to the message broker (or event bus)

    - subscribers register with that broker
    - broker performs filtering and routing
    - broker may perform prioritization
- **Data Distribution Service topology**
    - publishers and subscribers share metadata about each other (via IP multicast)
    - they cache this information locally and route message based on the discovery in the shared cognizance
- **Implementation**
    - Filtering forms
        - Topic-based - messages are published to topics, subscribers receive all messages of the topic the are subscribed to
        - Content-based - messages are delivered to subscribers, if attributes or content matches
        - Hybrid
    - Registration time
        - Build time - e.g. hard coded subscribers handling user commands
        - Initialization time - e.g. frameworks using configurations to register subscribers during system initialization
        - Runtime - e.g. database triggers, mailing lists, ...
- Cons
    - Message delivery issues
    - limited scalability of the pub/sub network
    - as the system grows, the message volume flow will slow down

Asynchronous messaging

- Asynchronous communication between heterogeneous components

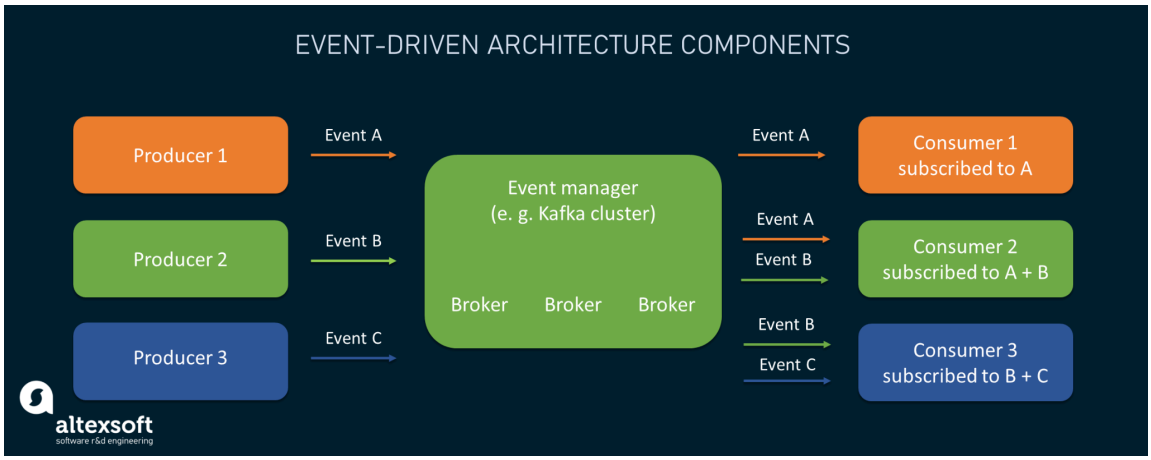
- Tightly coupled with Publish-Subscribe architecture



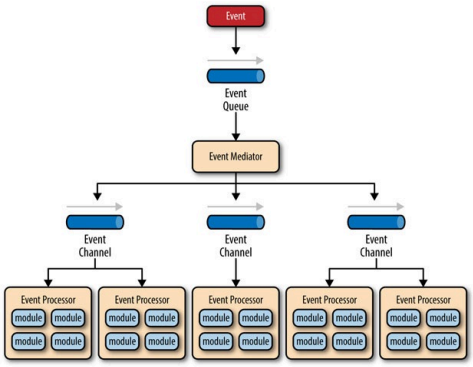
- Pros
    - store, route and transform messages
    - we can monitor and tune performance of the messaging system - interoperability, reliability, security, scalability, performance
    - solves connectivity issues and sender/receiver failure problems
- Cons
    - Bus/Broker maintenance

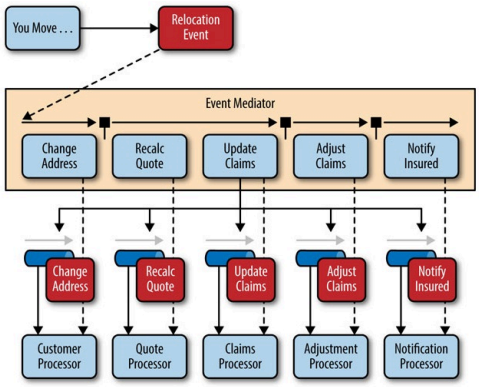
Event-Driven architecture

- Architecture for distributed systems
- Promotes asynchronous message communication

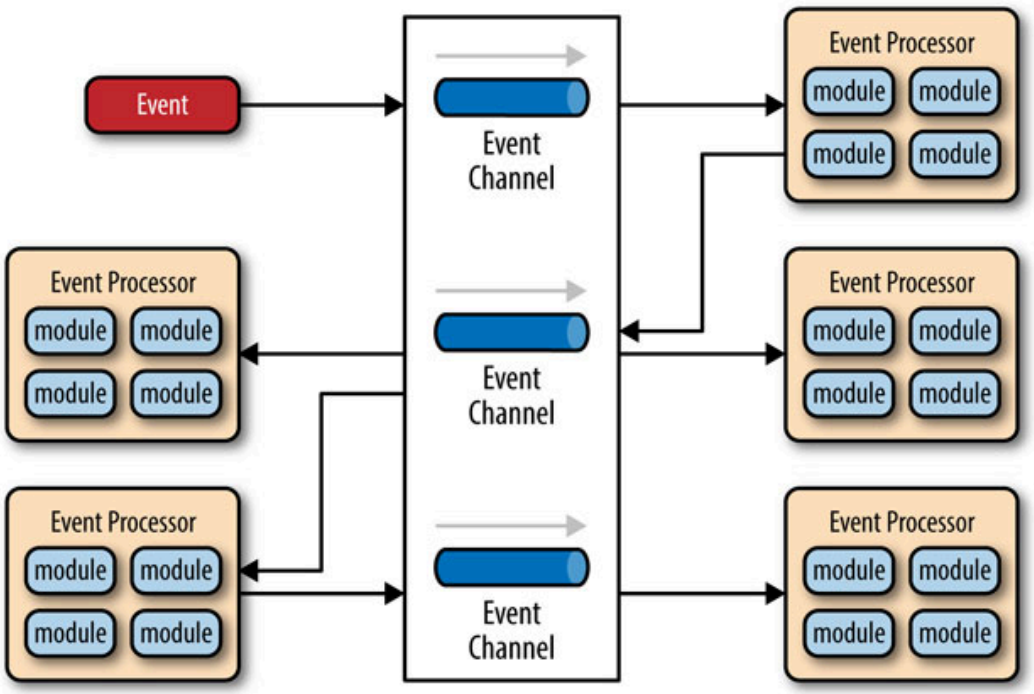


- Simplifies horizontal scalability
- Can complement Service Oriented Architecture
- **Mediator topology**
    - For events that have multiple steps and require some level of orchestration to process the event





- **Broker topology**
  - For simple event processing flow

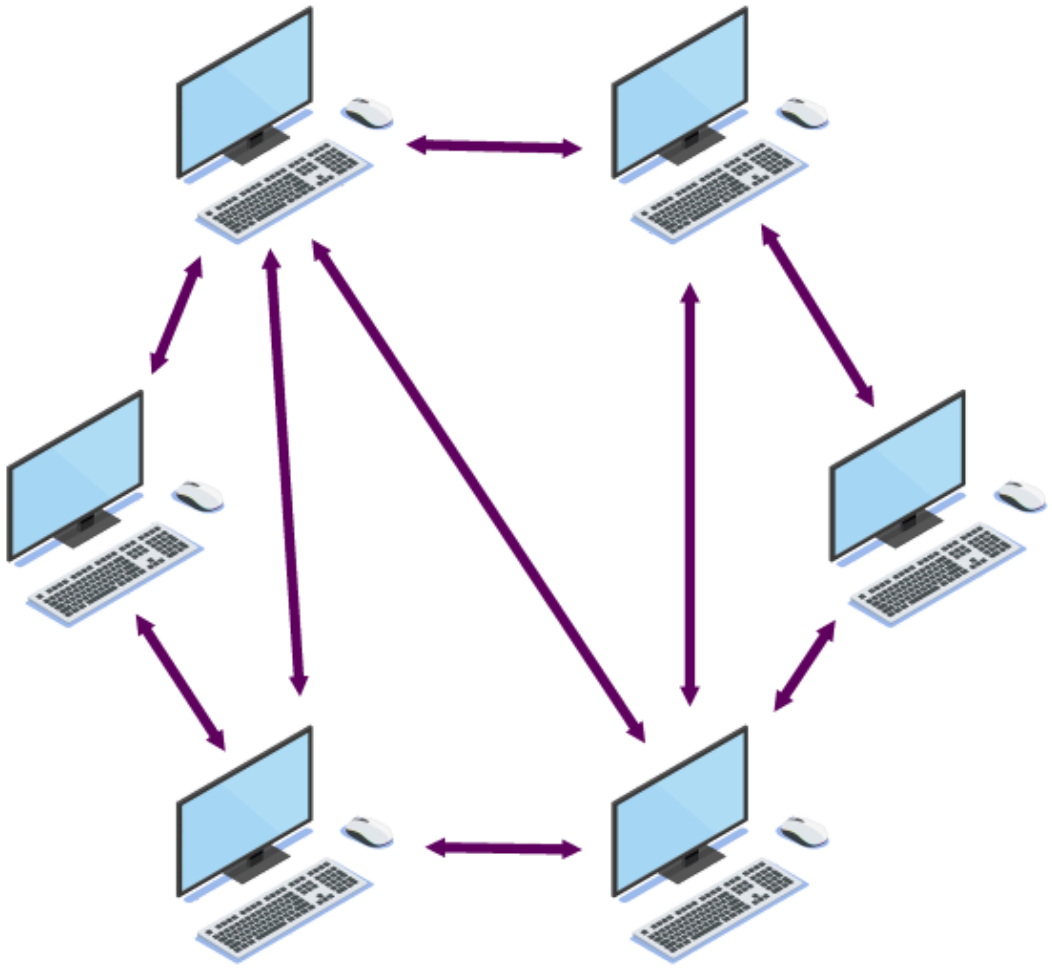


- **Event**
  - Represents a change of state etc…
  - Event header and Event Body
- Pros
  - Any component can be introduced into the system
  - Components can be easily replaced
  - Well distributed
  - High availability
  - Extreme loose coupling

- Cons
    - Tightly coupled to the semantics of the underlying event schema and values

Peer-to-peer architecture (P2P)

- Decentralized network architecture where each participant (peer) shares part of their resources with other peers
- Resource - processing power, storage space, network bandwidth, ...
- Doesn't require central server, each peer acts as client-server
- Allows for direct sharing of files, data and resources

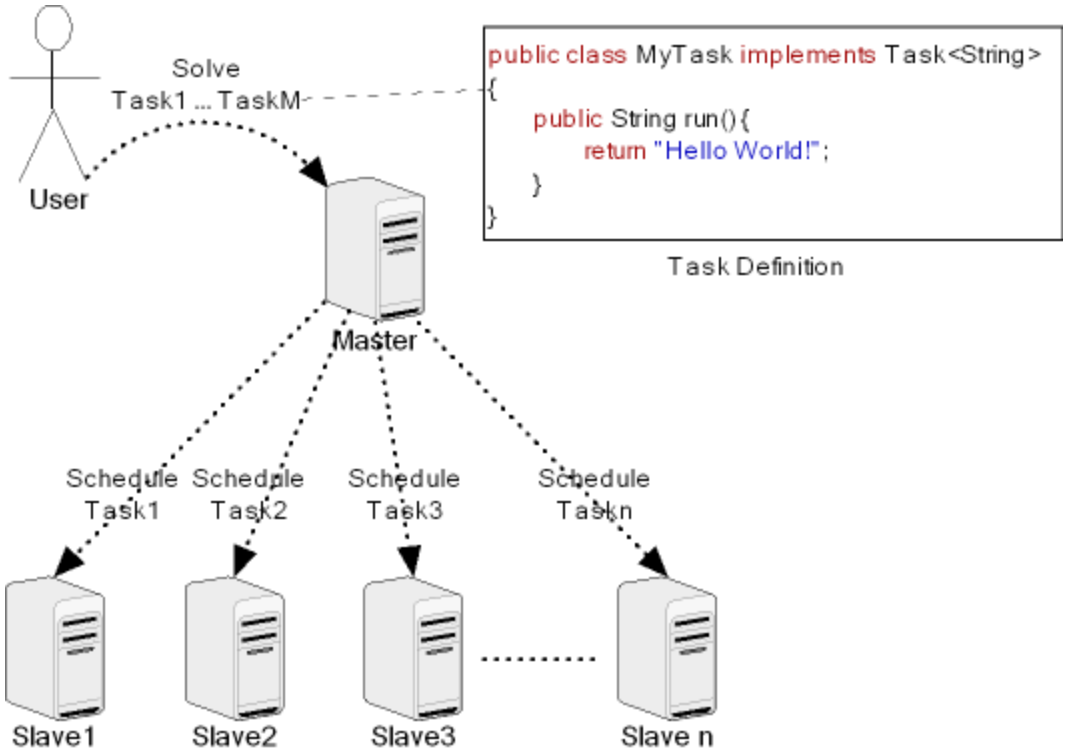


-
- File sharing, Blockchain, Grid computing, Communication, Content distribution (CDNs)
- Pros

    - Scalability
    - Robustness, no single point of failure
    - Cost-Effective
    - Resource sharing
- Cons
    - Security, no central authority to monitor data transfer
    - Inconsistent performance, depending on the peers
    - Legal and Ethical issues
    - Complex management

Master-Slave architecture

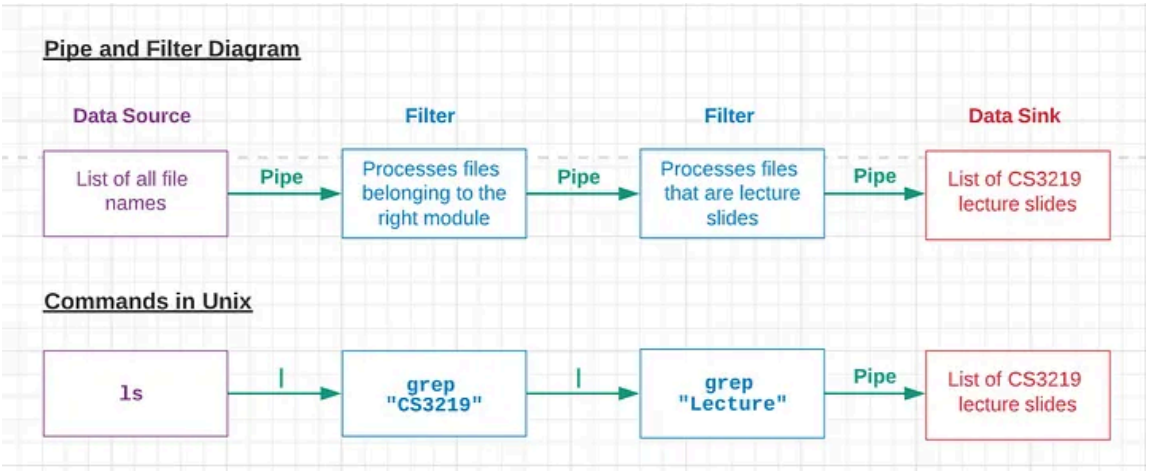
- Master component controls multiple slave components
- Slave components perform actions and return results back to master
- Commonly used to simplify complex processes, distribute workloads, manage tasks efficiently



- DB replication, Load balancing, Parallel processing, Distributed computing
- Pros:
    - Efficiency, distributing tasks

    - Scalability, adding multiple slaves
    - Fault tolerance, fail of one slave doesn't matter
- Cons:
    - Single point of failure - master
    - Complexity, managing synchronization and communication
    - Scalability limits - master is a bottleneck

Pipe and filter architecture

- Filters (processing elements) are connected by pipes (channels)
- Filters process data and pass them to next filter in a pipeline
- Similar to pipes and filters in Unix command line
- Commonly used for streaming data processing
- 

- Data processing, Streaming applications, Compiler, Image processing
- Pros:
    - Modularity
    - Reusability, filters can be reused
    - Parallel processing, filters can process data in parallel
    - Flexibility, easy to add/remove filters
- Cons:
    - Performance overhead - filters are bottleneck
    - Managing state across filters is complex
    - Difficult debugging

    - Limited interactivity, not suited for real-time applications, data must pass through the entire pipeline

Rule-based architecture

- Provide a means of codifying the problem-solving knowhow of experts
- Behavior of the system is defined by a set of rules
- Commonly used in expert system, decision support systems, business rule management system
- Components
    - knowledge base - pseudo-code to be executed
    - interpretation engine - Rule interpreter
    - control state of the engine - Rule and data element selector
    - current state of the program - working memory
- Rules are stored in Rule-base (knowledge base)
    - IF condition THEN action

```
// Rule interface
interface LoanApprovalRule {
    boolean evaluate(ApplicationData application);
}

// Concrete Rule 1: CreditScoreRule
class CreditScoreRule implements LoanApprovalRule {
    @Override
    public boolean evaluate(ApplicationData application) {
        return application.getCreditScore() > 650;
    }
}

// Concrete Rule 2: IncomeRule
class IncomeRule implements LoanApprovalRule {
    @Override
    public boolean evaluate(ApplicationData application) {
        return application.getIncome() > 50000;
    }
}

// Concrete Rule 3: EmploymentRule
```

```
class EmploymentRule implements LoanApprovalRule {
    @Override
    public boolean evaluate(ApplicationData application) {
        return application.getEmploymentYears() >= 2;
    }
}

// ApplicationData class representing the data in a loan application
class ApplicationData {
    private int creditScore;
    private double income;
    private int employmentYears;

    // constructor and getters

    // Additional methods to set data
}

// RuleEngine class to manage and evaluate rules
class RuleEngine {
    private List<LoanApprovalRule> rules;

    public RuleEngine(List<LoanApprovalRule> rules) {
        this.rules = rules;
    }

    public boolean processLoanApplication(ApplicationData application) {
        for (LoanApprovalRule rule : rules) {
            if (!rule.evaluate(application)) {
                return false; // Application does not meet a rule, reject
the loan
            }
        }
        return true; // All rules passed, approve the loan
    }
}

// Client code
public class RuleBasedExample {
    public static void main(String[] args) {
        // Define rules
        List<LoanApprovalRule> rules = Arrays.asList(
```

```
                new CreditScoreRule(),
                new IncomeRule(),
                new EmploymentRule()
        );

        // Create a rule engine
        RuleEngine ruleEngine = new RuleEngine(rules);

        // Create an application
        ApplicationData application = new ApplicationData();
        application.setCreditScore(700);
        application.setIncome(60000);
        application.setEmploymentYears(3);

        // Process the loan application
        boolean approvalStatus =
ruleEngine.processLoanApplication(application);

        // Display the result
        if (approvalStatus) {
            System.out.println("Loan approved!");
        } else {
            System.out.println("Loan rejected.");
        }
    }
}
```

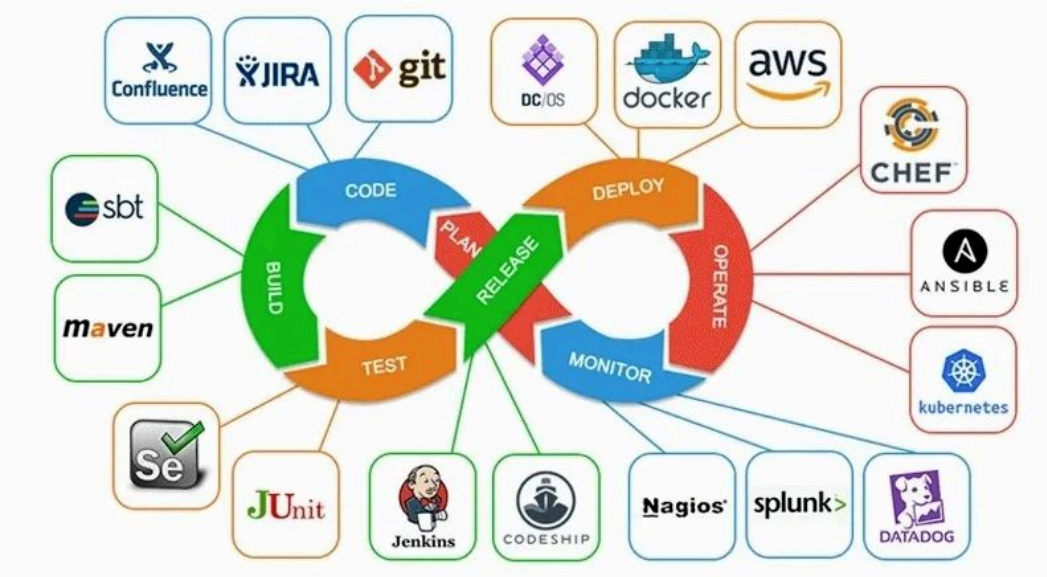
Blackboard architecture

- Complex problem decomposed into distinct independent components - agents
- Agents work in parallel, each contributing specialized knowledge to a shared global repository - Blackboard
- Components
    - Blackboard - structured global memory
        - agents deposit and retrieve information
    - Knowledge sources (Agents) - specialized modules
    - Control component (Scheduler) - selects, configures and executes modules, orchestrates the collaboration
- Useful for problems for which no deterministic solution strategies are known

- Multiple specialized systems assemble their knowledge to build a possibly partial or approximate solution
- Domains
    - speech recognition
    - sonar signal interpretation
- Stackoverflow is basically a Blackboard, developers are agents sharing their knowledge about the problem
- Blackboard was outperformed by statistical pattern recognition techniques (Hidden Markov Models), therefore usually abandoned architecture

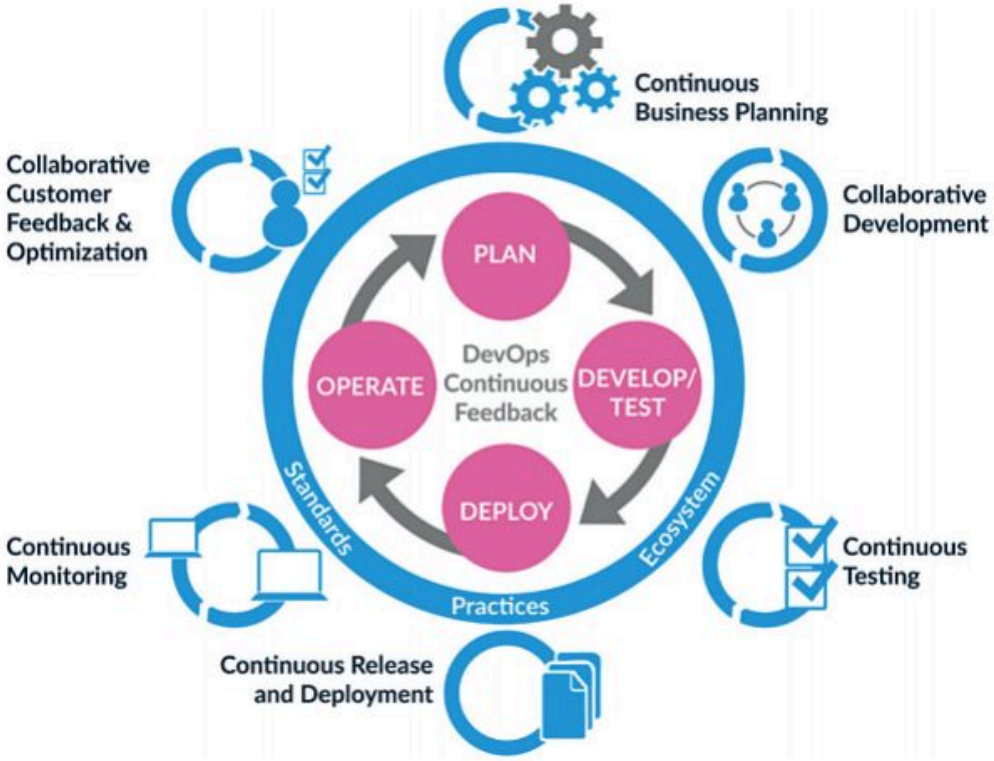
DevOps architecture (principles)

- A mindset or a culture for better collaboration, communication and continuous improvement when it comes to releasing a better SW in the most efficient way
- Inspired by Agile development
- **Principles**
    - Collaboration - enables communication between cross-functional teams, user feedback, sprints, developers, ...
    - Automation - automating processes = eliminating resource waste, reducing effort etc... automate as much as possible, testing, deployment, reduces human error, saves time



    - Iteration - frequent commits, rapid releases, revision control, ...

    - Continuous improvement - Continuous process based on CI/CD principles
- **The DevOps Workflow**
    - Continuous planning
    - Continuous integration (collaborative development)
    - Continuous testing
    - Continuous delivery
    - Continuous deployment
    - Continuous monitoring

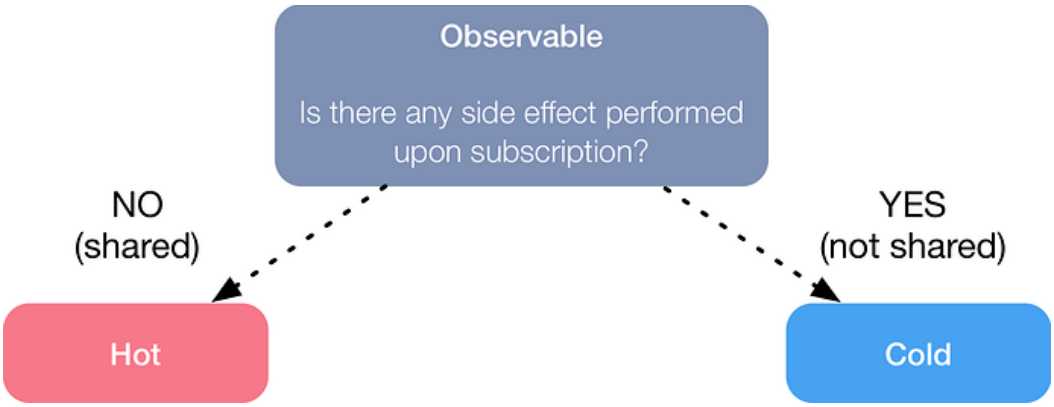


-

Reactive programming

- Declarative programming paradigm concerned with data streams
- Based on Observer pattern

- observables are either Hot or Cold



- **Cold observables** - provide the same data right after subscription for all subscribers
- **Hot observables** - provide data in time and whoever isn't subscribed yet, won't get the data

Functional programming

TODO

Map Reduce

TODO