Unit - 4

Behavioral Patterns

- Behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns.
- Behavioral patterns influence how state and behavior flow through a system.

Mediator: Provides a unified interface to a set of interfaces in a subsystem
Memento: Provides the ability to restore an object to its previous state (rollback)
Observer: also known as Publish/Subscribe or Event Listener. Objects register to observe an event that may be raised
by another object
State: A clean way for an object to partially change its type at runtime
Strategy: Algorithms can be selected on the fly
Template Method: Describes the program skeleton of a program
Visitor: A way to separate an algorithm from an object

List of common Behavioral Design Patterns

Chain of Responsibility Pattern

Intent:

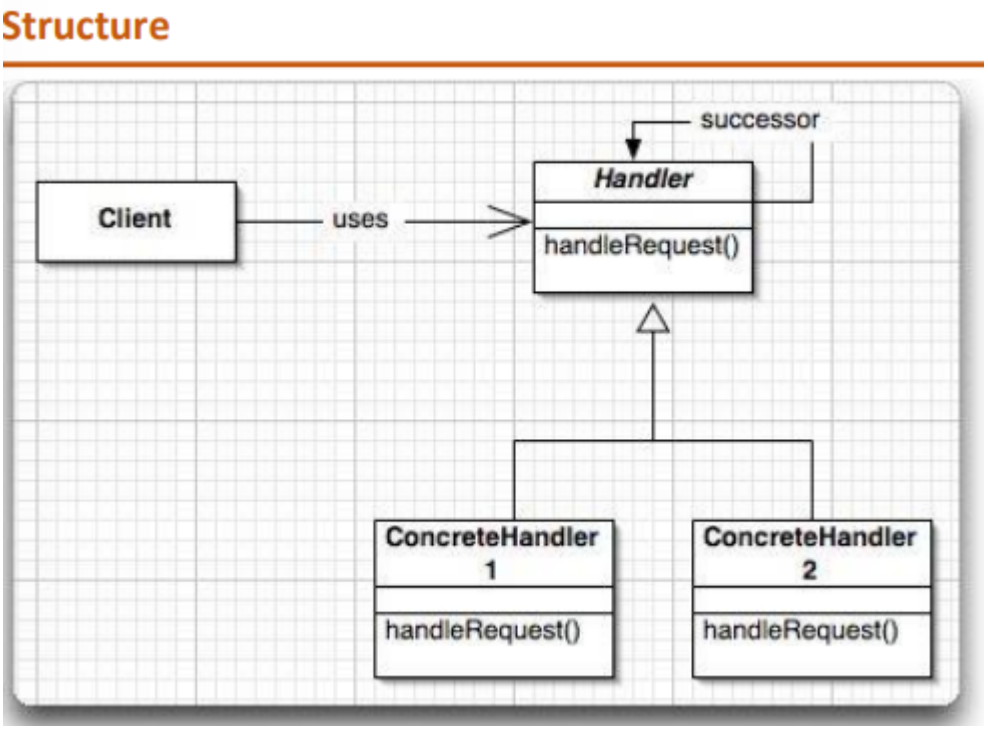
- Avoid coupling sender-of-request to its receiver by giving more than one object a chance to handle request.
- Chain receiving objects and pass request along until an object handles it.

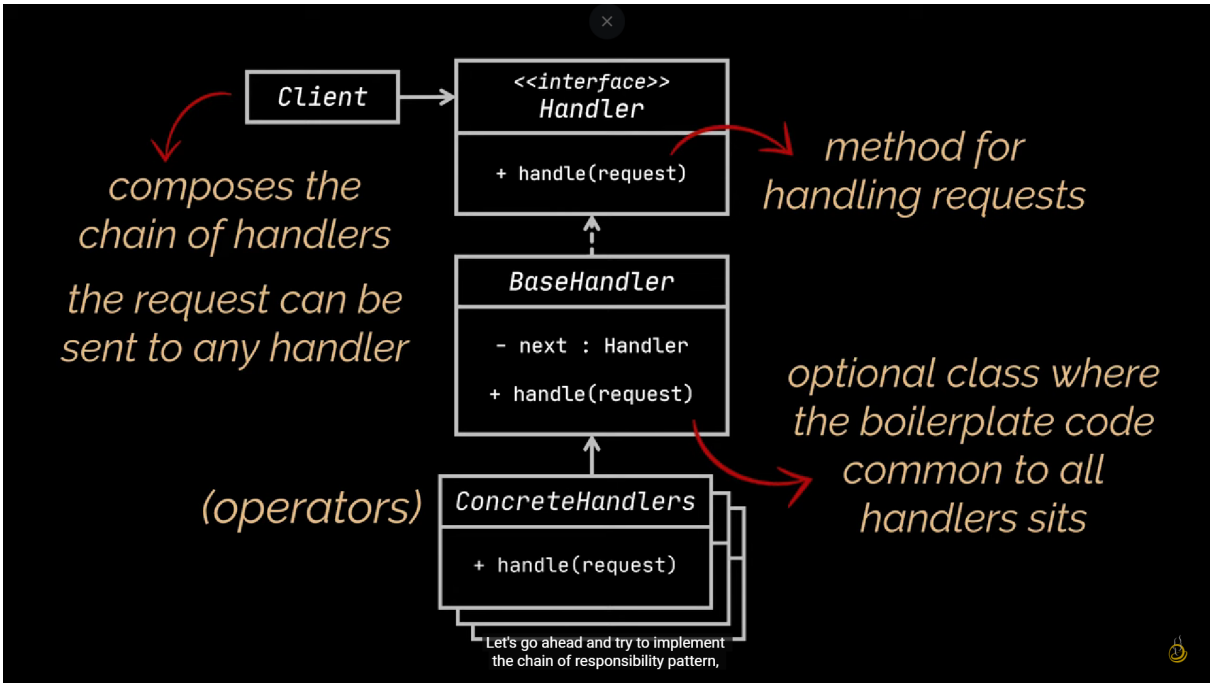
Motivation:

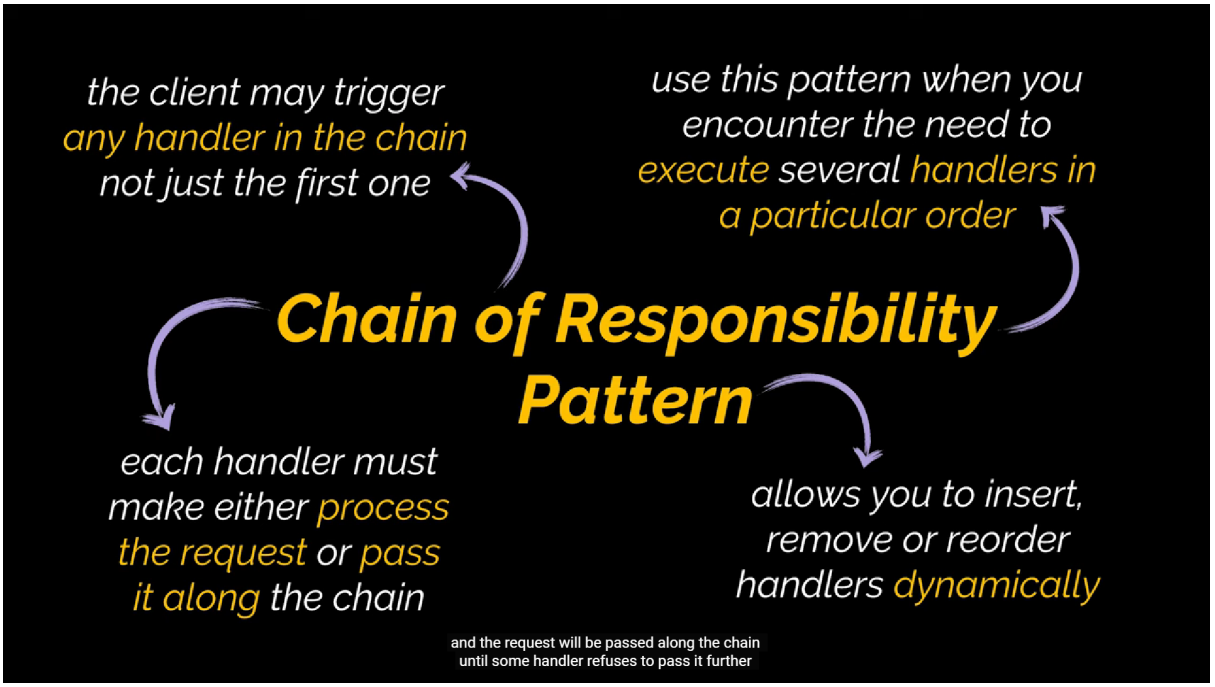
- The Chain of Responsibility is intended to promote loose coupling between the sender of a request and its receiver by giving more than one object an opportunity to handle the request.
- The receiving objects are chained and pass the request along the chain until one of the objects handles it.
- The set of potential request handler objects and the order in which these objects form the chain can be decided dynamically at runtime by the client depending on the current state of the application.

Usage:

- Used to manage algorithms, relationships, and responsibilities between objects within a system.







Participants:

Handler:

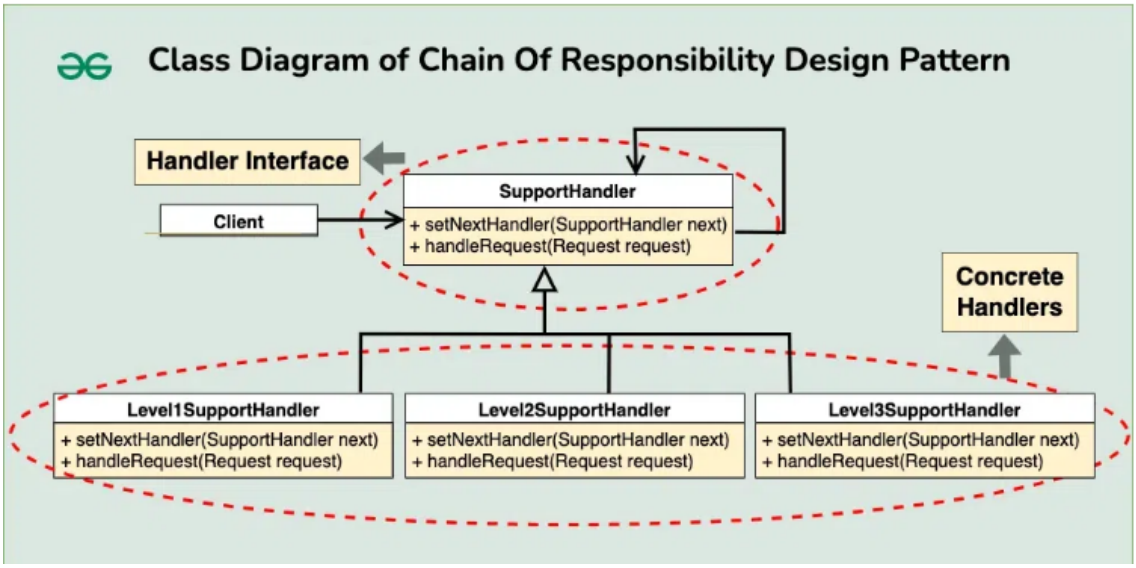
- This can be an interface which will primarily receive the request and dispatches the request to a chain of handlers. It has a reference to only the first handler in the chain and does not know anything about the rest of the handlers.

ConcreteHandler:

- Handles requests it is responsible for.
- Can access its successor.
- If it does not handle the request, forwards the request to its successor.

Client:

- Initiates the request to a ConcreteHandlerObject on the chain.

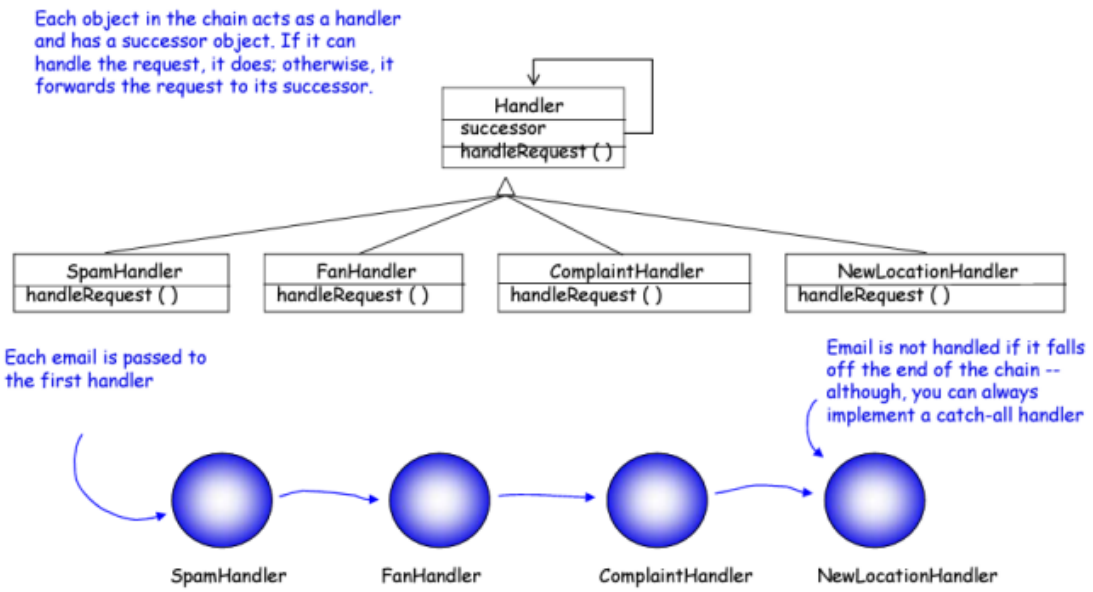


Scenario:
An enterprise has been receiving more email than they can handle. The enterprise receives 4 types of email:

1. Fan mail with compliments.
2. Complaints.
3. Requests for new features.
4. Spam.

Task:
The enterprise has already developed AI detectors that can classify emails as fan mail, complaints, requests, or spam. You need to create a design that can use these detectors to handle incoming email.

Example – Scenario 1:
Reduced Coupling:



- Objects are free from knowing what object handles the request.
  Added Flexibility in assigning responsibilities to objects:
- Can change the chain at runtime.
- Can subclass for special handlers.
  Receipt is guaranteed:
- Request could fall off the chain.
- Request could be dropped with a bad chain.

How to Implement

1. Declare the handler interface and describe the signature of a method for handling requests.

   Decide how the client will pass the request data into the method. The most flexible way is to convert the request into an object and pass it to the handling method as an argument.

2. To eliminate duplicate boilerplate code in concrete handlers, it might be worth creating an abstract base handler class, derived from the handler interface.

   This class should have a field for storing a reference to the next handler in the chain. Consider making the class immutable. However, if you plan to modify chains at runtime, you need to define a setter for altering the value of the reference field.

You can also implement the convenient default behavior for the handling method, which is to forward the request to the next object unless there's none left. Concrete handlers will be able to use this behavior by calling the parent method.

3. One by one create concrete handler subclasses and implement their handling methods. Each handler should make two decisions when receiving a request:
    - Whether it'll process the request.
    - Whether it'll pass the request along the chain.
4. The client may either assemble chains on its own or receive pre-built chains from other objects. In the latter case, you must implement some factory classes to build chains according to the configuration or environment settings.
5. The client may trigger any handler in the chain, not just the first one. The request will be passed along the chain until some handler refuses to pass it further or until it reaches the end of the chain.
6. Due to the dynamic nature of the chain, the client should be ready to handle the following scenarios:
    - The chain may consist of a single link.
    - Some requests may not reach the end of the chain.
    - Others may reach the end of the chain unhandled.

Use Chain of Responsibility when:

- More than one object may handle a request, and the handler isn't known a prior.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

Applicability:
Implementation – Example Scenario 2

Consider the transaction approval process in a company. Suppose we want to approve transactions based on certain conditions. For instance, in the

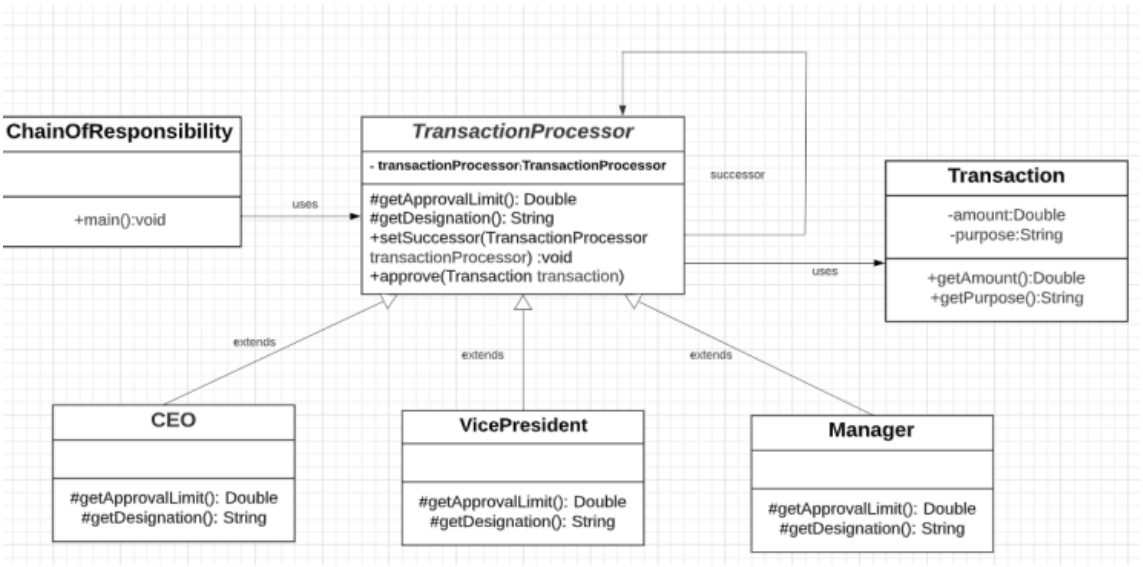
transaction approval application, the user enters transaction details, and this transaction needs to be processed by any one of the higher-level employees in the company.

- All transactions less than 1,00,000 can be approved by a manager.
- Transactions less than 10,00,000 can be approved by a vice president.
- Transactions less than 25,00,000 can be approved by the CEO.

The transaction approval behavior can be implemented with a simple if-else conditions, by checking if the amount is less than 1,00,000, or else if the amount is less than 10,00,000, and so on. But this approach is static, meaning we cannot change these conditions dynamically. The Chain of Responsibility design pattern can be used in this situation.

In the transaction processing application, the approval process acts like a chain of responsibilities. First, the manager acts on the transaction. If the amount is higher than his approval limit, then the transaction is transferred to the vice president, and so on.

We need to design processing and command objects. Based on the above example, processing objects are employees like Manager, Vice President because they process the transaction by approving, and the command object is the transaction. Once processing objects are created, we need to chain them together like manager → vice president → CEO and pass the transaction at the beginning of the chain. The transaction continues to flow in the chain until it reaches the employee whose limit allows them to approve it.



```
// Handler Interface
interface SupportHandler {
    void handleRequest(Request request);
    void setNextHandler(SupportHandler nextHandler);
}

// Concrete Handlers
class Level1SupportHandler implements SupportHandler {
    private SupportHandler nextHandler;

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

    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.BASIC) {
            System.out.println("Level 1 Support handled the r
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        } else {
            System.out.println("Request cannot be handled.");
        }
    }
}

class Level2SupportHandler implements SupportHandler {
    private SupportHandler nextHandler;

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

    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.INTERMEDIATE) {
            System.out.println("Level 2 Support handled the r
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        } else {
```

```
            System.out.println("Request cannot be handled.");
        }
    }
}

class Level3SupportHandler implements SupportHandler {
    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.CRITICAL) {
            System.out.println("Level 3 Support handled the r
        } else {
            System.out.println("Request cannot be handled.");
        }
    }

    public void setNextHandler(SupportHandler nextHandler) {
        // No next handler for Level 3
    }
}

// Request Class
class Request {
    private Priority priority;

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

    public Priority getPriority() {
        return priority;
    }
}

// Priority Enum
enum Priority {
    BASIC, INTERMEDIATE, CRITICAL
}

// Main Class
```

```
public class Main {
    public static void main(String[] args) {
        SupportHandler level1Handler = new Level1SupportHandl
        SupportHandler level2Handler = new Level2SupportHandl
        SupportHandler level3Handler = new Level3SupportHandl

        level1Handler.setNextHandler(level2Handler);
        level2Handler.setNextHandler(level3Handler);

        Request request1 = new Request(Priority.BASIC);
        Request request2 = new Request(Priority.INTERMEDIATE)
        Request request3 = new Request(Priority.CRITICAL);

        level1Handler.handleRequest(request1);
        level1Handler.handleRequest(request2);
        level1Handler.handleRequest(request3);
    }
}
```

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

Benefits:

- Decouples the sender of the request and its receivers.
- Simplifies your object because it doesn't have to know the chain's structure and keep direct reference to its members.
- Allows you to add or remove responsibilities dynamically by changing the members or the order of the chain.

Uses:

- Commonly used in Windows systems to handle events like mouse clicks and keyboard events.
- In Java, it is used in handling a chain of Exceptions.

Drawbacks:

- Execution of the request isn't guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage).

- Can be hard to observe the runtime characteristics and debug.

Command Design Pattern

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

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

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

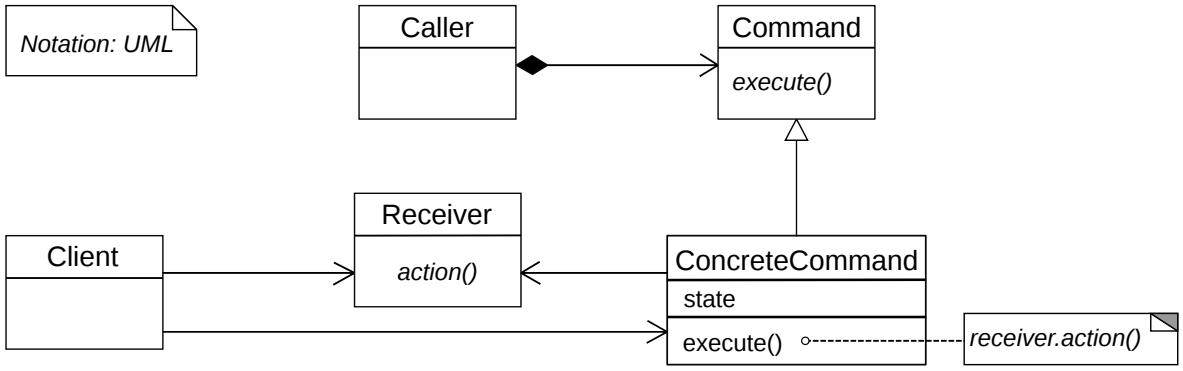
Intent:

- Encapsulate a request in an object.
- Allow the parameterization of clients with different requests.
- Allow saving the requests in a queue.

Motivation:

- Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.
- For example, user interface toolkits include objects like buttons and menus that carry out a request in response to user input. But the toolkit can't implement the request explicitly in the button or menu because only applications that use the toolkit know what should be done on which object. As toolkit designers, we have no way of knowing the receiver of the request or the operations that will carry it out. The Command pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object.

Structure:



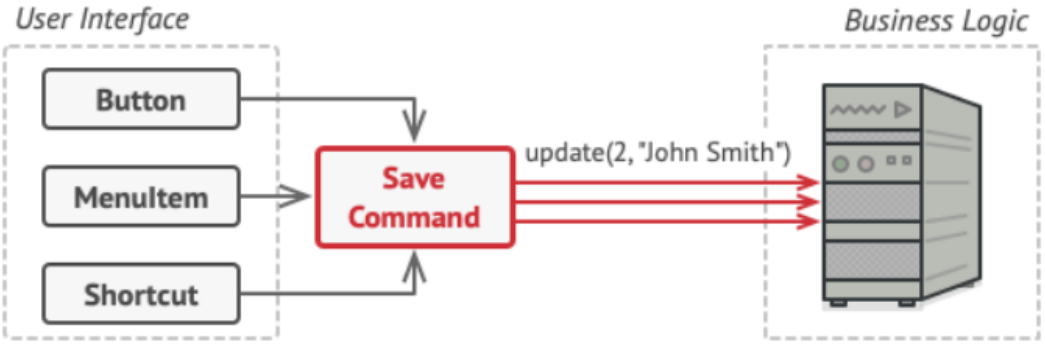
Participants:

- **Command:** Declares an interface for executing an operation.
- **ConcreteCommand:**
  - Defines a binding between a Receiver object and an action.
  - Implements execute() by invoking a corresponding operation on Receiver.
- **Client (Application):** Creates a Command object and sets its Receiver.
- **Invoker:** Asks the Command to carry out a request.
- **Receiver:** Knows how to perform the operation associated with a request. Can be any class.

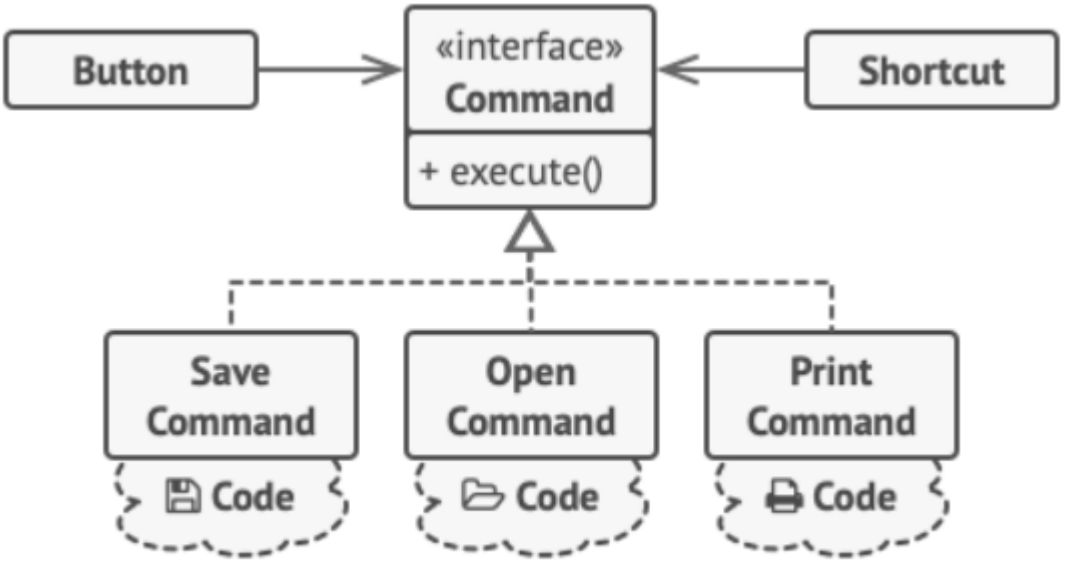
Collaborations:

- The Client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The invoker issues a request by calling execute() on the command. When commands are undoable, ConcreteCommand stores state for undoing prior to invoking execute().
- The ConcreteCommand object invokes operations on its receiver to carry out the request.

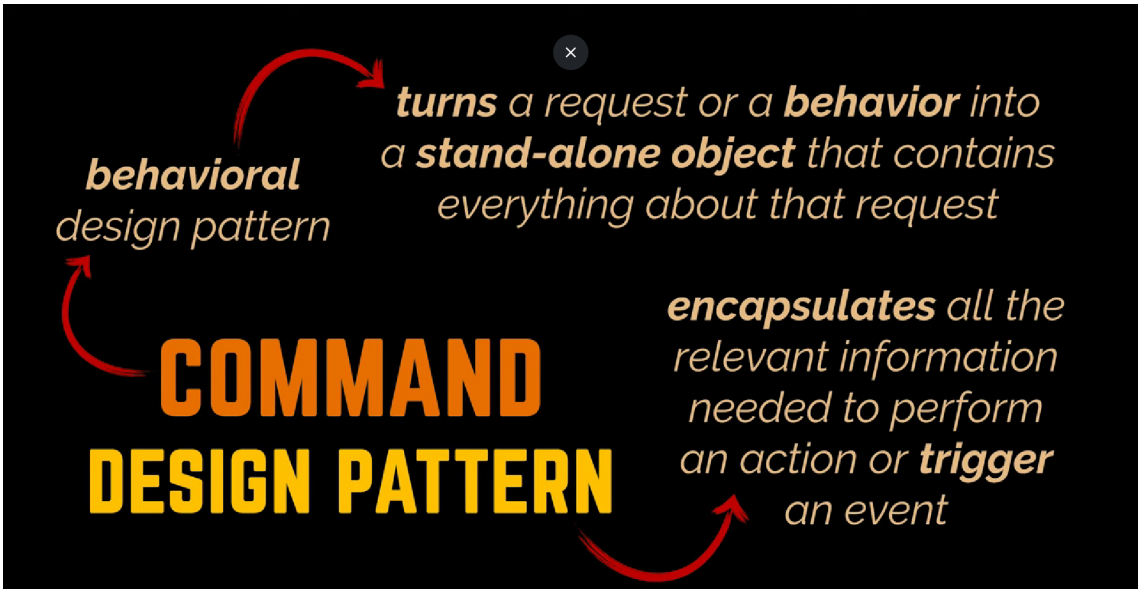
The GUI objects may access the business logic objects directly.

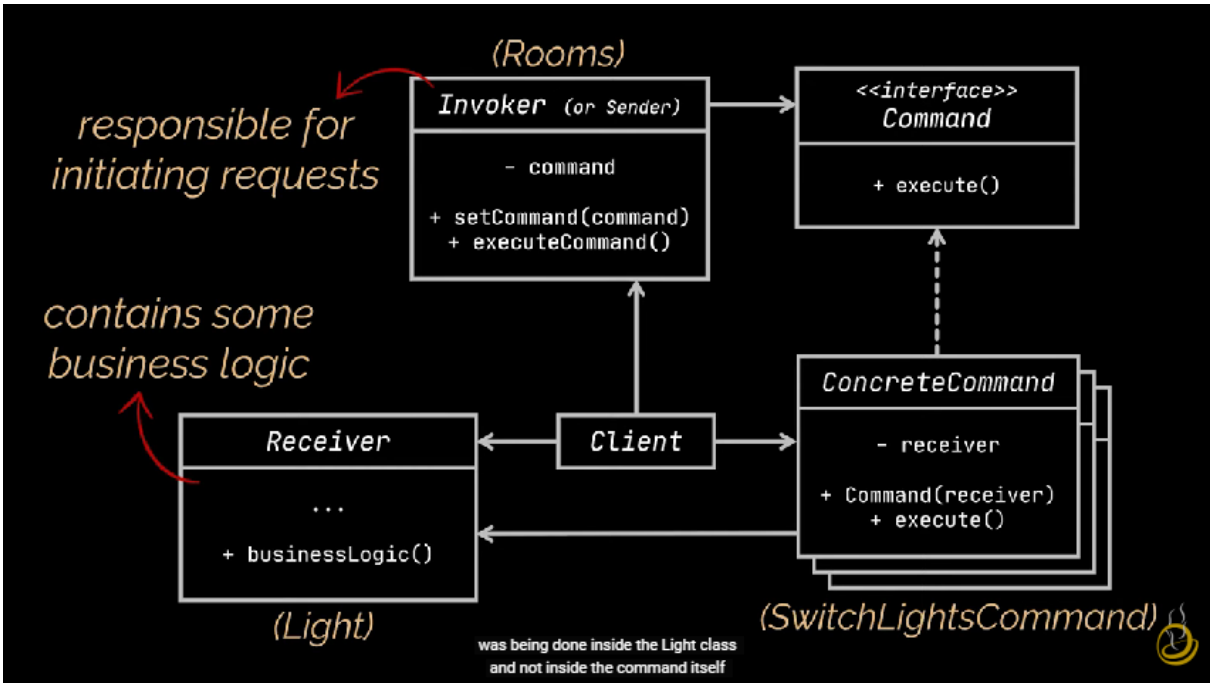


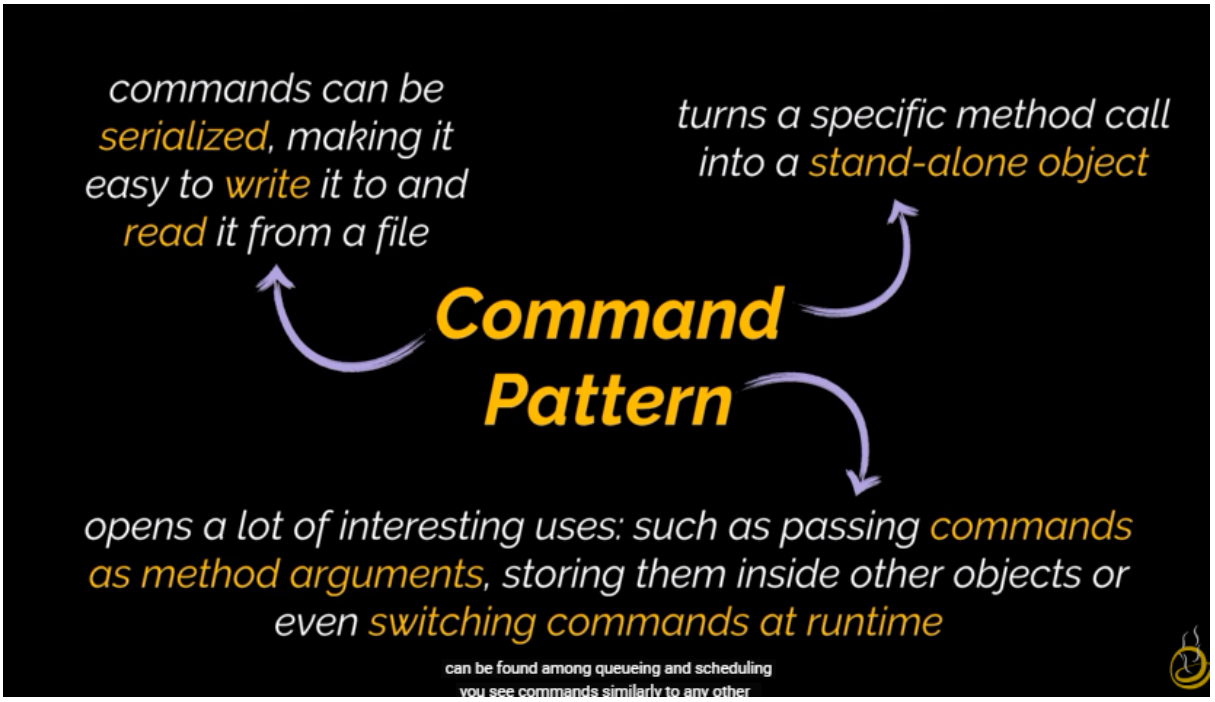
Accessing the business logic layer via a command.

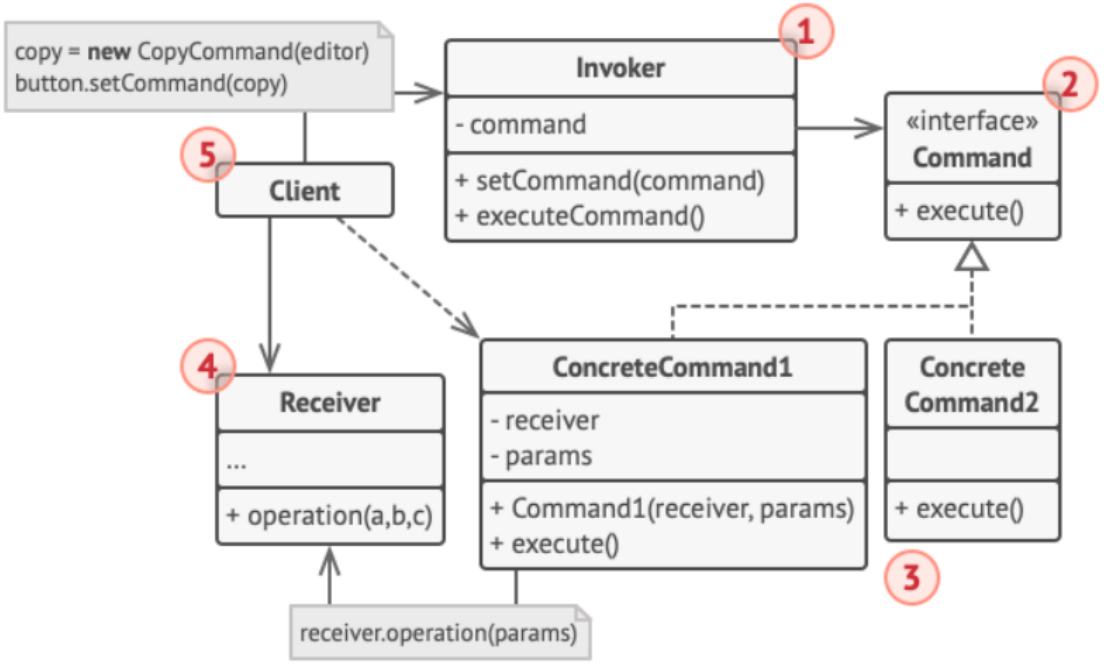


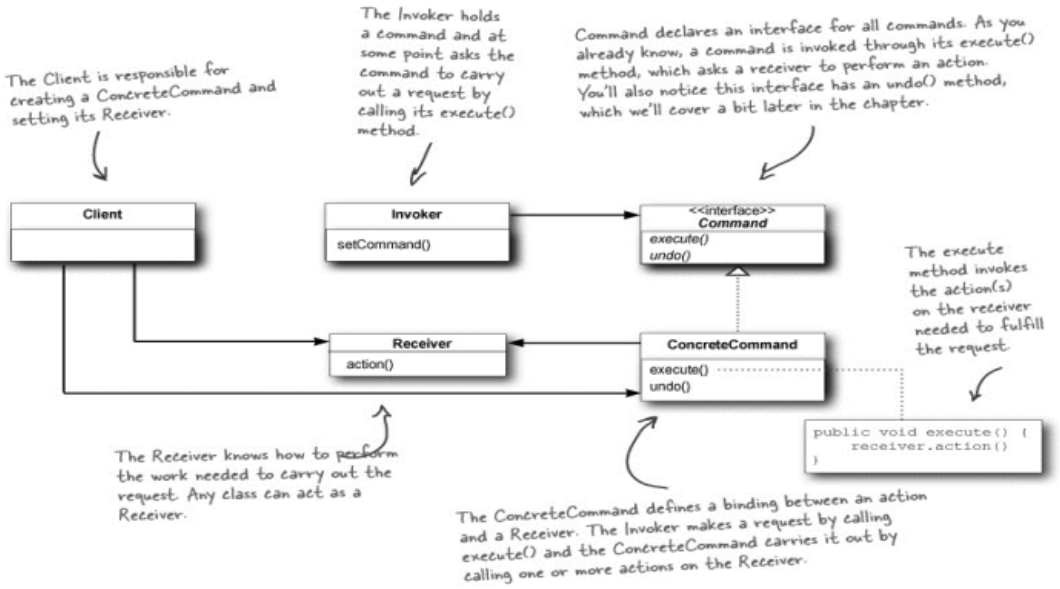
The GUI objects delegate the work to commands.


turns a request or a behavior into a stand-alone object that contains everything about that request
behavioral design pattern
COMMAND
DESIGN PATTERN
encapsulates all the relevant information needed to perform an action or trigger an event


(Rooms)
responsible for initiating requests
Invoker (or Sender)
- command
+ setCommand(command)
+ executeCommand()
<<interface>>
Command
+ execute()
contains some business logic
ConcreteCommand
- receiver
+ Command(receiver)
+ execute()
Receiver
...
+ businessLogic()
Client
(Light)
(SwitchLightsCommand)
was being done inside the Light class
and not inside the command itself


commands can be serialized, making it easy to write it to and read it from a file
turns a specific method call into a stand-alone object
Command Pattern
opens a lot of interesting uses: such as passing commands as method arguments, storing them inside other objects or even switching commands at runtime
can be found among queueing and scheduling
you see commands similarly to any other


copy = new CopyCommand(editor)
button.setCommand(copy)
Invoker
- command
+ setCommand(command)
+ executeCommand()
«interface»
Command
+ execute()
Client
Receiver
...
+ operation(a,b,c)
ConcreteCommand1
- receiver
- params
+ Command1(receiver, params)
+ execute()
Concrete
Command2
+ execute()
receiver.operation(params)
1
2
3
4
5



1. The **Sender** class (aka *invoker*) is responsible for initiating requests. This class must have a field for storing a reference to a command object. The sender triggers that command instead of sending the request directly to the receiver. Note that the sender isn't responsible for creating the command object. Usually, it gets a pre-created command from the client via the constructor.
2. The **Command** interface usually declares just a single method for executing the command.
3. **Concrete Commands** implement various kinds of requests. A concrete command isn't supposed to perform the work on its own, but rather to pass the call to one of the business logic objects. However, for the sake of simplifying the code, these classes can be merged.

   Parameters required to execute a method on a receiving object can be declared as fields in the concrete command. You can make command objects immutable by only allowing the initialization of these fields via the constructor.
4. The **Receiver** class contains some business logic. Almost any object may act as a receiver. Most commands only handle the details of how a request is passed to the receiver, while the receiver itself does the actual work.
5. The **Client** creates and configures concrete command objects. The client must pass all of the request parameters, including a receiver instance, into the command's constructor. After that, the resulting command may be associated with one or multiple senders.

Example Implementation in Java:

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

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

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

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

// Receiver
class Light {
    public void turnOn() {
        System.out.println("Light is on");
    }

    public void turnOff() {
        System.out.println("Light is off");
    }
}

// Invoker
class RemoteControl {
    private Command command;

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

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

// Client
public class Main {
    public static void main(String[] args) {
        Light light = new Light();
        Command lightOnCommand = new LightOnCommand(light);

        RemoteControl remote = new RemoteControl();
        remote.setCommand(lightOnCommand);

        remote.pressButton(); // Light is on
    }
}
```

In this example:

- `Command` is the interface for all concrete command classes.
- `LightOnCommand` is a concrete command that turns on the light.
- `Light` is the receiver of the command.
- `RemoteControl` is the invoker of the command.
- `Main` is the client that creates and sets the command to the invoker. When the button is pressed, the invoker executes the command.

Benefits:

- Decouples the classes that invoke the operation from the object that knows how to execute the operation.
- Allows you to create a sequence of commands by providing a queue system.
- Extensions to add a new command are easy and can be done without changing the existing code.

- You can also define a rollback system with the Command pattern. For example, in the Wizard example, we could write a rollback method.

Drawbacks:

- There is a high number of classes and objects working together to achieve a goal. Application developers need to be careful when developing these classes correctly.
- Every individual command is a ConcreteCommand class that increases the volume of classes for implementation and maintenance.

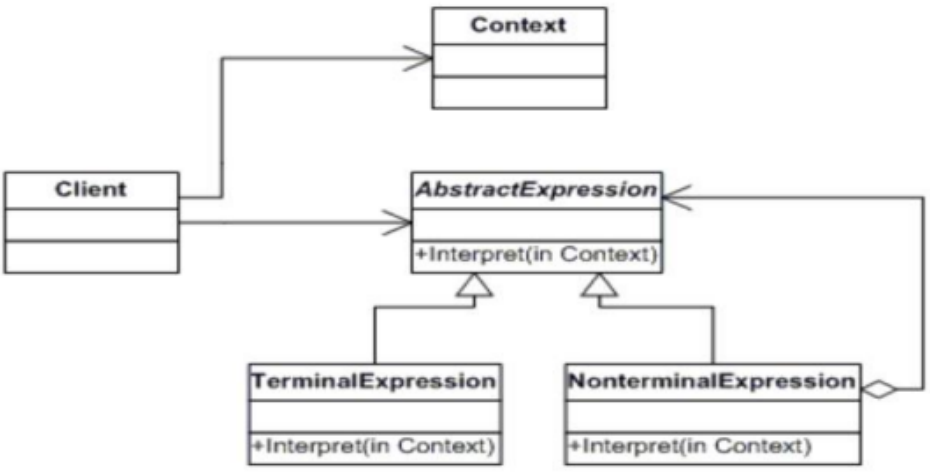
Use the Command pattern when you want to parametrize objects with operations.

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

Use the Command pattern when you want to implement reversible operations.

Behavioural Patterns – Interpreter

Interpreter Design Pattern:



Intent:

- Provide a way to interpret and execute expressions in a domain-specific language.

- Break down complex expressions into smaller components that can be easily interpreted.
- Create parsers and compilers for the defined language.
- Build scripting languages and other domain-specific languages.
- Provide a flexible and extensible way to define and interpret expressions in the language.
- Support the implementation of rules that govern the behavior of the language.
- Facilitate the creation of abstract syntax trees to represent expressions in the language.
- Allow for the creation of interpreters that can handle different types of expressions and operators.

Motivation:

- Define a domain-specific language: It allows developers to define a language that is specific to a particular domain or problem. This language can be tailored to meet the needs of a specific application, making it easier to write expressive and concise code.
- Interpret and execute expressions dynamically: It provides a way to interpret and execute expressions at runtime. This is useful when dealing with complex expressions that are difficult or impossible to evaluate statically.
- Separate concerns: It separates the concerns of defining a language and interpreting expressions in that language. This makes it easier to modify the language and add new expressions without affecting the interpreter code.
- Build parsers and compilers: The Interpreter pattern can be used to build parsers and compilers for the defined language. This allows the language to be used in a wide range of applications and contexts.
- Facilitate communication between different components: The Interpreter pattern can be used to facilitate communication between different components in a system. By defining a common language, different components can exchange messages and work together more easily.

Design Solution – Participants:

AbstractExpression (Expression):

- It declares an interface for executing an operation.

TerminalExpression (ThousandExpression, HundredExpression, TenExpression, OneExpression):

- It implements an Interpret operation associated with terminal symbols in the grammar.
- An instance is required for every terminal symbol in the sentence.

NonterminalExpression (not used):

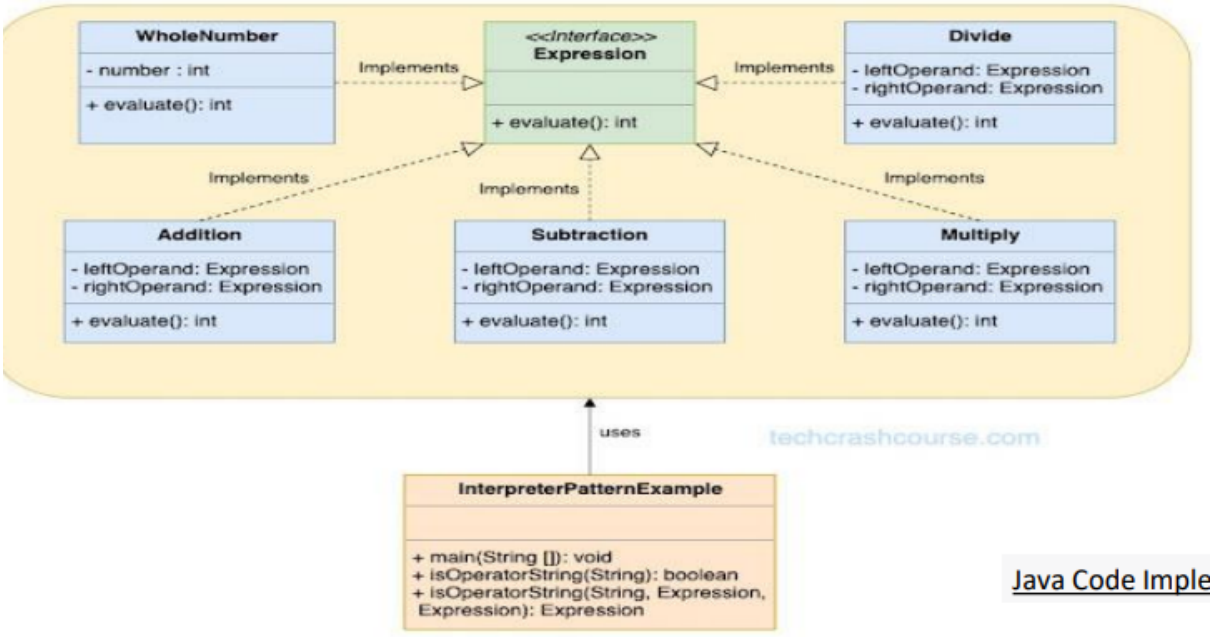
- One such class is required for every rule R ::= R1R2...Rn in the grammar and maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
- Implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.

Context (Context):

- It contains information that is global to the interpreter.

Client (InterpreterApp):

- It builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines.
- The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes.
- It invokes the Interpret operation.



Java Code Imple

Applications:

- **Regular Expressions:** Regular expressions are a powerful tool for matching patterns in strings. An interpreter can be used to parse and execute regular expressions, making it easier to build complex matching patterns.
- **SQL Queries:** SQL queries can be complex and difficult to write. An interpreter can be used to parse and execute SQL queries, allowing developers to build complex queries without having to understand the underlying implementation details.
- **Mathematical Expressions:** Mathematical expressions can be difficult to parse and evaluate. An interpreter can be used to parse and evaluate mathematical expressions, allowing you to build complex calculations and formulas.
- **Programming Languages:** Interpreters are commonly used in programming languages to execute code. They can be used to interpret scripts, run-time code, or compile code on-the-fly.

Advantages:

- **Flexibility:** The Interpreter pattern provides a flexible way to implement complex grammars or expressions. It allows you to define your own language and parse sentences in that language.
- **Extensibility:** The Interpreter pattern makes it easy to add new functionality or behavior to your application. You can simply add new expressions or

modify existing ones to change the behavior of your program.

- **Separation of Concerns:** The Interpreter pattern separates the parsing and execution logic, making it easier to maintain and extend your code.

Disadvantages:

- **Complexity:** The Interpreter pattern can add complexity to your code, especially when dealing with complex grammars or expressions.
- **Performance:** The Interpreter pattern can be slower than other approaches, such as compiling or using a parser generator, since it requires interpreting the expressions at run-time.
- **Maintenance:** The Interpreter pattern can make your code more difficult to maintain, especially if the language or expressions change frequently.

Consequence:

- The Interpreter pattern is a powerful tool that can make your code more flexible and extensible.
- However, it also comes with some trade-offs in terms of complexity, performance, and maintenance. It's important to consider these factors when deciding whether to use the Interpreter pattern in your application.

Behavioral Patterns – Iterator

Iterator Design Pattern:

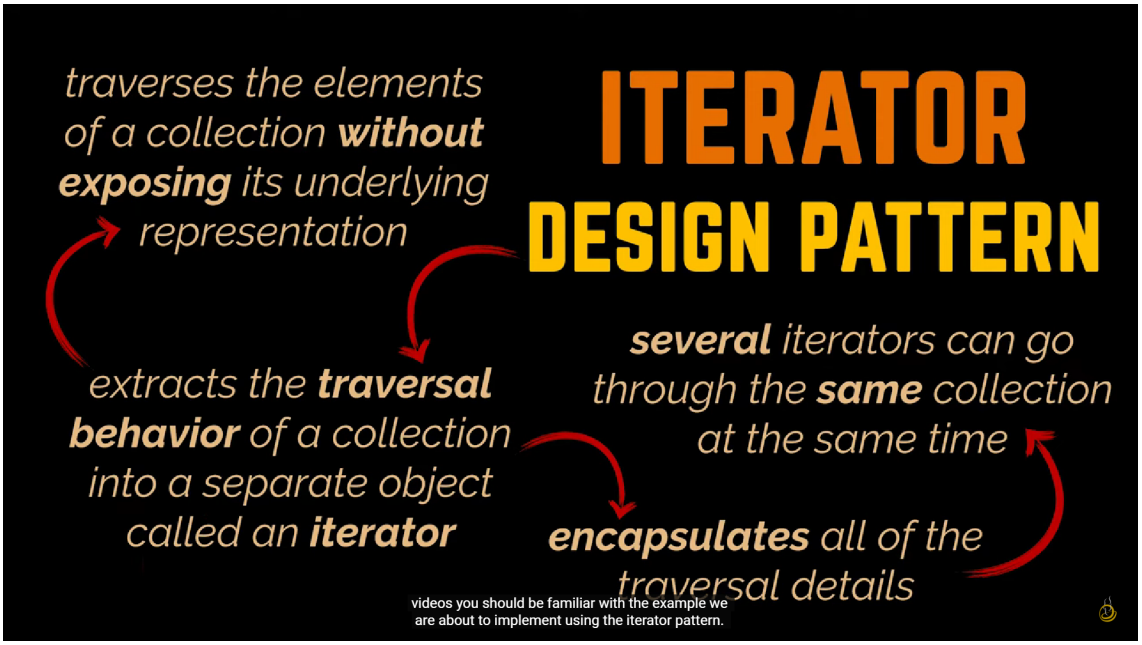
Intent:

- The Iterator design pattern provides a way to access the elements of an aggregate object sequentially, without exposing its underlying implementation.
- It offers a uniform way to access the elements of a collection, such as a list, set, or map, without having to know how the collection is implemented.
- The intent of using an iterator is to provide a standardized way of accessing and manipulating collections of data in a simple, efficient, and flexible manner.
- It is useful for implementing algorithms that require sequential access to data, such as sorting, searching, and filtering. By using an iterator, you can

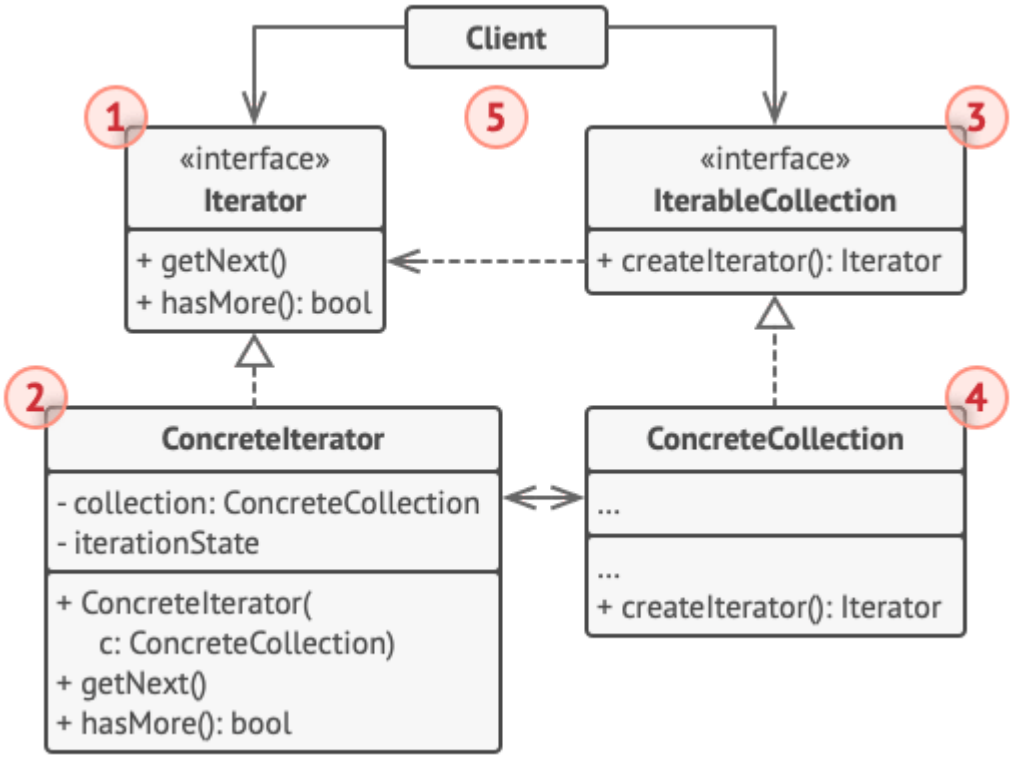
easily apply these algorithms to any collection of data, regardless of its specific implementation or data structure.

Motivation:

- **Encapsulation:** The Iterator pattern encapsulates the traversal logic of a collection of objects in a separate object, allowing the collection to change its internal representation without affecting the code that uses it.
- **Abstraction:** The Iterator pattern abstracts the process of iterating over a collection of objects by providing a standard interface that can be used by algorithms and clients without having to know the specific implementation details of the collection.
- **Separation of concerns:** The Iterator pattern separates the concerns of managing a collection of objects from the concerns of using the objects in the collection. This makes it easier to write modular, reusable, and maintainable code.
- **Multiple traversal support:** The Iterator pattern allows traversing a collection of objects multiple times without having to reset the traversal state of the collection.
- **Lazy evaluation:** The Iterator pattern supports lazy evaluation of collections, where the elements of the collection are computed or retrieved only when they are needed. This can be especially useful for working with large datasets or data streams.



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



Design Solution – Participants:

Iterator (AbstractIterator):

- Defines an interface for accessing and traversing elements.

ConcreteIterator (Iterator):

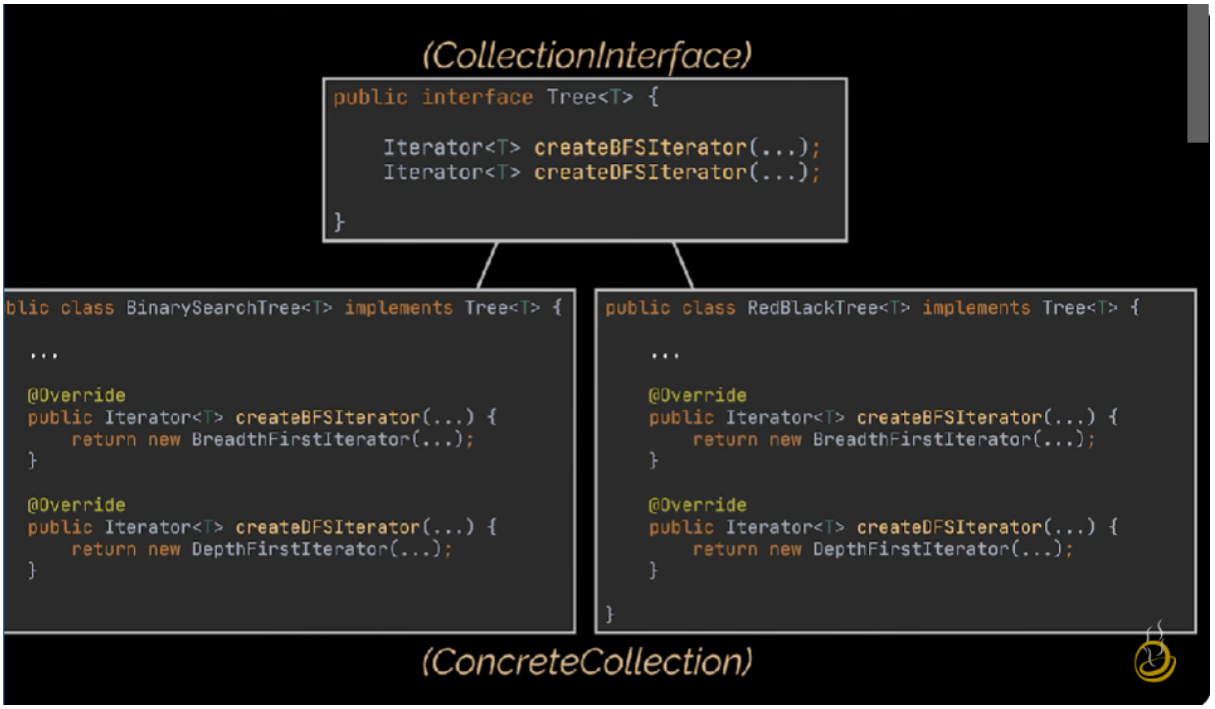
- Implements the Iterator interface.
- Keeps track of the current position in the traversal of the aggregate.

Aggregate (AbstractCollection):

- Defines an interface for creating an Iterator object.

ConcreteAggregate (Collection):

- Implements the Iterator creation interface to return an instance of the proper ConcreteIterator.



Consequence:

Consequences of using the Iterator design pattern are:

1. **Improved code flexibility:** The Iterator pattern allows clients to iterate over collections in different ways without changing the underlying collection's structure. This improves code flexibility and makes it easier to add new iteration methods in the future.

2. **Better encapsulation:** By providing a separate object to handle iteration, the internal details of the collection are hidden from the client code. This improves encapsulation and helps to prevent code from becoming tightly coupled.
3. **Simplified client code:** The Iterator pattern simplifies the client code by providing a standardized way to access collection elements. This reduces the complexity of the client code and makes it easier to read and maintain.
4. **Performance overhead:** The Iterator pattern may introduce a performance overhead, as the iteration process involves additional method calls and object creations. However, this overhead is typically negligible for small collections.

Overall, the Iterator design pattern is a useful tool for improving the flexibility, encapsulation, and maintainability of code that involves iterating over collections.

Simplifying complex data structures:

- The Iterator pattern is often used to simplify complex data structures, such as trees and graphs, by providing a way to traverse them in a standardized manner.

Implementing lazy evaluation:

- The Iterator pattern can be used to implement lazy evaluation of data, where elements are only computed or retrieved when they are needed, rather than loading the entire collection into memory at once.

Providing a standard interface:

- The Iterator pattern provides a standard interface for accessing and manipulating collections of data, making it easier to write algorithms that work with different types of collections.

Supporting multiple traversals:

- The Iterator pattern makes it possible to traverse a collection of objects multiple times without having to reset the traversal state of the collection.

Applicability:

- Accessing elements in a collection.
- Simplifying complex data structures.

- Implementing lazy evaluation.
- Supporting multiple traversals.
- Providing a standard interface.

Advantages:

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

Disadvantages:

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

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

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

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

Structural Pattern

Introduction to Structural Design Pattern:

- Structural design patterns are concerned with how classes and objects can be composed to form larger structures.
- These patterns simplify the structure by identifying the relationships between classes and objects.

- They focus on how classes inherit from each other and how they are composed from other classes.
- Structural class patterns use inheritance to compose interfaces or implementations.
- They explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

Types of Structural Design Patterns:

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Adapter Pattern

Motivation:

- The adapter pattern adapts between classes and objects, acting as a bridge between them.
- Just like any adapter in the real world, it is used to provide an interface between two objects. For example, adapters for power supplies or camera memory cards.
- Sometimes, a toolkit class designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

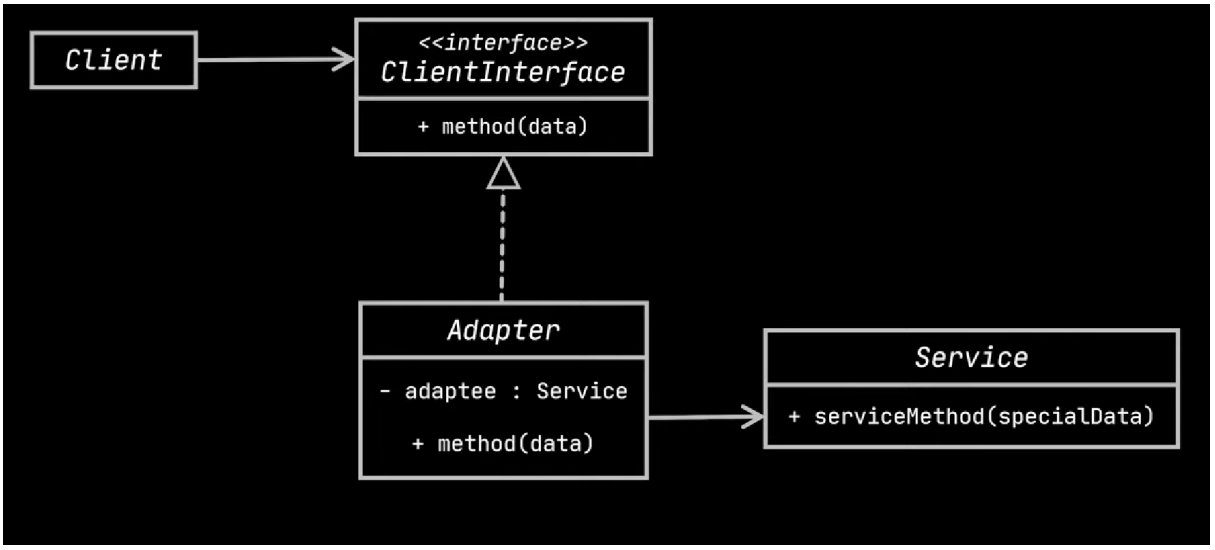
Advantages of Adapter Pattern:

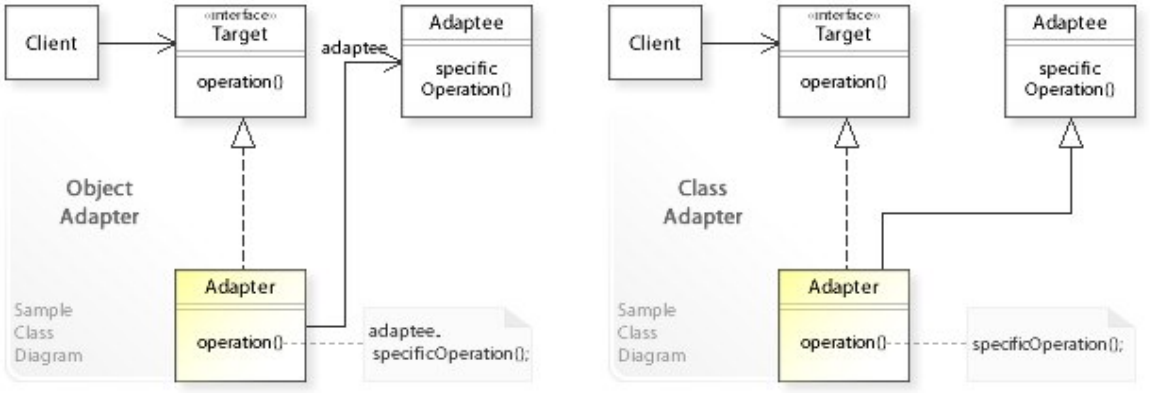
- It allows two or more previously incompatible objects to interact.
- It allows the reusability of existing functionality.

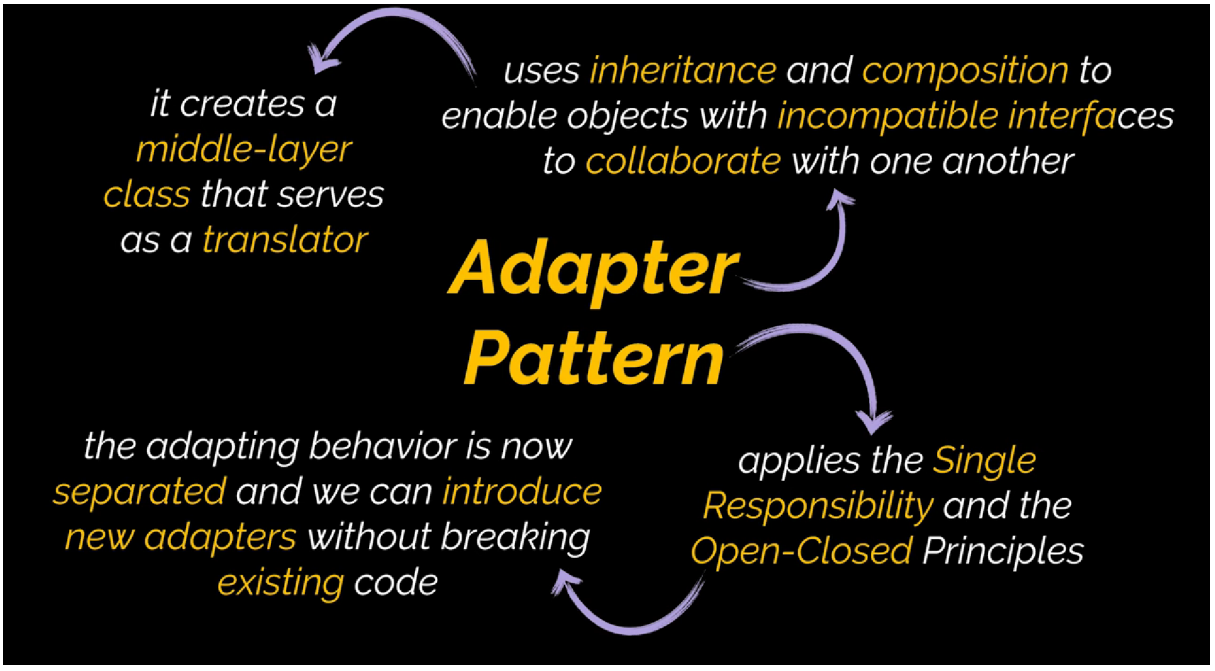
Intent:

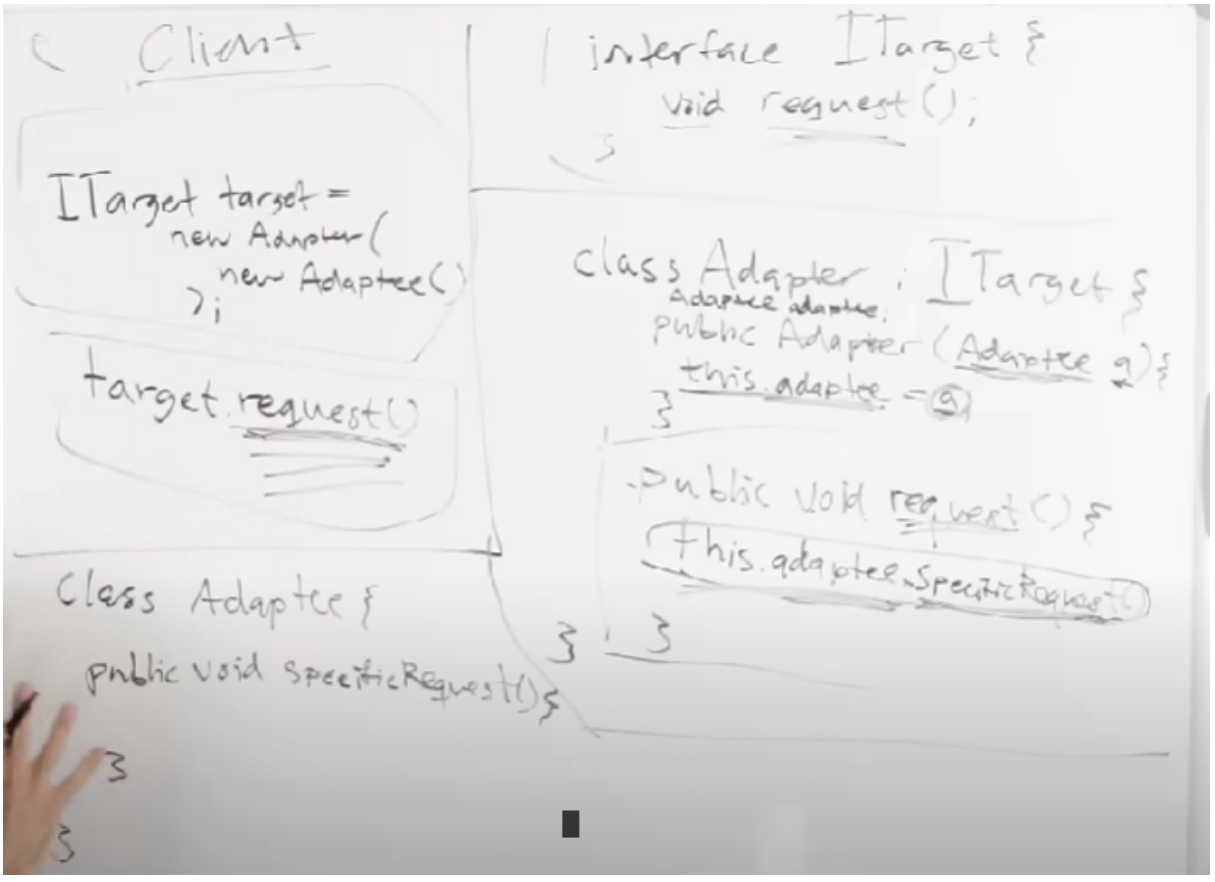
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- The Adapter Pattern converts the interface of a class into another interface that a client wants, providing the interface according to the client requirement while using the services of a class with a different interface.
- The Adapter Pattern is also known as Wrapper.

Adapter Pattern Implementation:









Classes/Objects Participating in Adapter Pattern:

1. **Target:** Defines the domain-specific interface that the client uses.
2. **Adapter:** Adapts the interface of the Adaptee to the Target interface.

3. **Adaptee:** Defines an existing interface that needs adapting.
4. **Client:** Collaborates with objects conforming to the Target interface.

Adapter class: This class is a wrapper class that implements the desired target interface and modifies the specific request available from the Adaptee class.

Sure, here are the problem statements for the Adapter pattern examples:

Example 1:
Problem Statement: We have a `MediaPlayer` interface and a concrete class `AudioPlayer` implementing the `MediaPlayer` interface. `AudioPlayer` can play mp3 format audio files by default. We are having another interface `AdvancedMediaPlayer` and concrete classes implementing the `AdvancedMediaPlayer` interface. These classes can play vlc and mp4 format files. We want to make `AudioPlayer` to play other formats as well.

```
// MediaPlayer.java
public interface MediaPlayer {
    public void play(String audioType, String fileName);
}

// AdvancedMediaPlayer.java
public interface AdvancedMediaPlayer {
    public void playVlc(String fileName);
    public void playMp4(String fileName);
}

// VlcPlayer.java
public class VlcPlayer implements AdvancedMediaPlayer{
    @Override
    public void playVlc(String fileName) {
        System.out.println("Playing vlc file. Name: "+ file
Name);
    }

    @Override
    public void playMp4(String fileName) {
        // do nothing
    }
}
```

```
// Mp4Player.java
public class Mp4Player implements AdvancedMediaPlayer{
    @Override
    public void playVlc(String fileName) {
        // do nothing
    }

    @Override
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file. Name: "+ file
Name);
    }
}

// MediaAdapter.java
public class MediaAdapter implements MediaPlayer {

    AdvancedMediaPlayer advancedMusicPlayer;

    public MediaAdapter(String audioType){
        if(audioType.equalsIgnoreCase("vlc") ){
            advancedMusicPlayer = new VlcPlayer();
        } else if (audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer = new Mp4Player();
        }
    }

    @Override
    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("vlc")){
            advancedMusicPlayer.playVlc(fileName);
        }else if(audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer.playMp4(fileName);
        }
    }
}
```

```
// AudioPlayer.java
public class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;

    @Override
    public void play(String audioType, String fileName) {

        //inbuilt support to play mp3 music files
        if(audioType.equalsIgnoreCase("mp3")){
            System.out.println("Playing mp3 file. Name: "+
fileName);
        }
        //mediaAdapter is providing support to play other f
ile formats
        else if(audioType.equalsIgnoreCase("vlc") || audioT
ype.equalsIgnoreCase("mp4")){
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        }
        else{
            System.out.println("Invalid media. "+ audioType
+ " format not supported");
        }
    }
}

// AdapterPatternDemo.java
public class AdapterPatternDemo {
    public static void main(String[] args) {
        AudioPlayer audioPlayer = new AudioPlayer();

        audioPlayer.play("mp3", "beyond the horizon.mp3");
        audioPlayer.play("mp4", "alone.mp4");
        audioPlayer.play("vlc", "far far away.vlc");
        audioPlayer.play("avi", "mind me.avi");
    }
}
```

Example 2:

Problem Statement: Suppose you have a `Bird` class with `fly()` and `makeSound()` methods. And also a `ToyDuck` class with `squeak()` method. Let's assume that you are short on `ToyDuck` objects and you would like to use `Bird` objects in their place. Birds have some similar functionality but implement a different interface, so we can't use them directly.

```
// Bird.java
public interface Bird {
    public void fly();
    public void makeSound();
}

// Sparrow.java
public class Sparrow implements Bird {
    @Override
    public void fly() {
        System.out.println("Flying");
    }

    @Override
    public void makeSound() {
        System.out.println("Chirp Chirp");
    }
}

// ToyDuck.java
public interface ToyDuck {
    public void squeak();
}

// PlasticToyDuck.java
public class PlasticToyDuck implements ToyDuck {
    @Override
    public void squeak() {
        System.out.println("Squeak");
    }
}
```

```
// BirdAdapter.java
public class BirdAdapter implements ToyDuck {

    Bird bird;

    public BirdAdapter(Bird bird){
        // we need reference to the object we
        // are adapting
        this.bird = bird;
    }

    @Override
    public void squeak() {
        // translate the methods appropriately
        bird.makeSound();
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        Sparrow sparrow = new Sparrow();
        ToyDuck toyDuck = new PlasticToyDuck();
        // Wrap a bird in a birdAdapter so that it
        // behaves like toy duck
        ToyDuck birdAdapter = new BirdAdapter(sparrow);

        System.out.println("Sparrow...");
        sparrow.fly();
        sparrow.makeSound();

        System.out.println("ToyDuck...");
        toyDuck.squeak();

        // toy duck behaving like a bird
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
```

```
    }
}
```

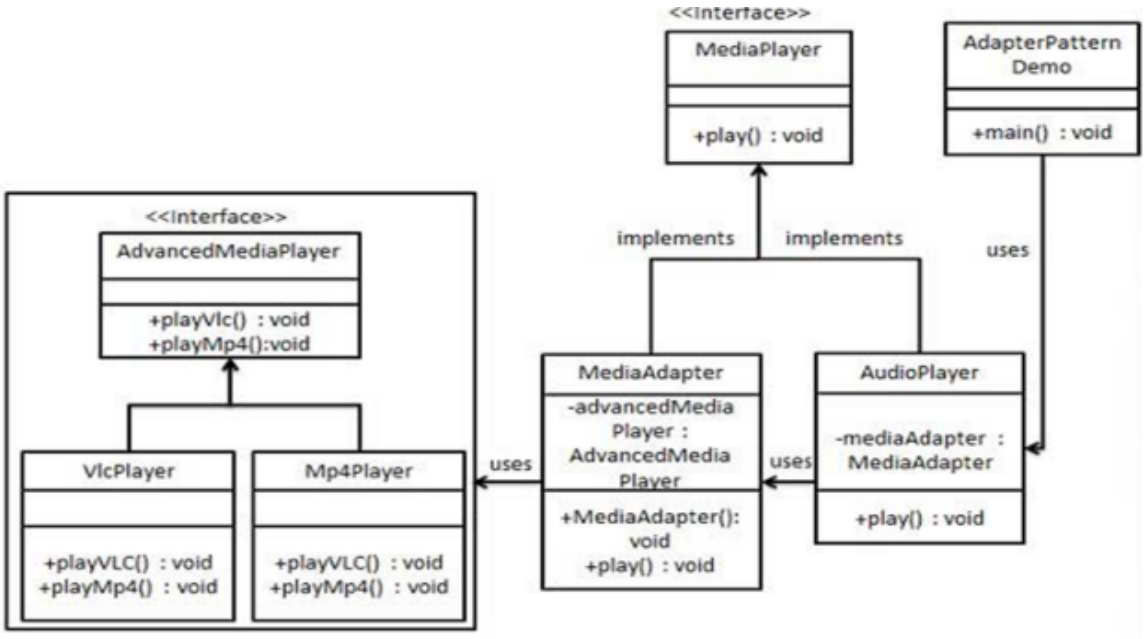
Example 3:

Problem Statement: Consider a scenario in which there is an app that's developed in the US which returns the top speed of luxury cars in miles per hour (MPH). Now we need to use the same app for our client in the UK that wants the same results but in kilometers per hour (km/h).

```
// USCarSpeedAdapter.java
public class USCarSpeedAdapter implements UKCarSpeed {
    USCarSpeed usCarSpeed;

    public USCarSpeedAdapter(USCarSpeed usCarSpeed) {
        this.usCarSpeed = usCarSpeed;
    }

    @Override
    public double getSpeed() {
        // convert mph to km/h
        return usCarSpeed.getSpeed() * 1.60934;
    }
}

// UKCarSpeed.java
public interface UKCarSpeed {
    public double getSpeed();
}

// USCarSpeed.java
public class USCarSpeed {
    public double getSpeed() {
        return 120; // speed in mph
    }
}

// Client.java
public class Client {
```

```
    public static void main(String[] args) {
        USCarSpeed usCarSpeed = new USCarSpeed();
        UKCarSpeed ukCarSpeed = new USCarSpeedAdapter(usCar
Speed);
        System.out.println("US Car Speed: " + usCarSpeed.ge
tSpeed() + " MPH");
        System.out.println("UK Car Speed: " + ukCarSpeed.ge
tSpeed() + " KM/H");
    }
}
```

The Adapter pattern is useful in several scenarios:



1. When you want to use an existing class, but its interface does not match the one you need.
2. When you want to create a reusable class that cooperates with unrelated or unforeseen classes, which don't necessarily have compatible interfaces.
3. When you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Here are some software examples of Adapter patterns:

- Wrappers used to adopt third-party libraries and frameworks. Most applications using third-party libraries use adapters as a middle layer between the application and the 3rd party library to decouple the application from the library.

Applicability:

- Accessing elements in a collection
- Simplifying complex data structures
- Implementing lazy evaluation
- Supporting multiple traversals
- Providing a standard interface

Structure:

- A class adapter uses multiple inheritance to adapt one interface to another.
- An object adapter relies on object composition based on delegation. Class adapters can be implemented in languages supporting multiple inheritance, while object adapters can be implemented in any object-oriented language.

Class and object adapters have different trade-offs:

Class Adapter:

- Adapts Adaptee to Target by committing to a concrete Adaptee class.
- Won't work when you want to adapt a class and all its subclasses.
- Allows Adapter to override some of Adaptee’s behavior, as Adapter is a subclass of Adaptee.
- Introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

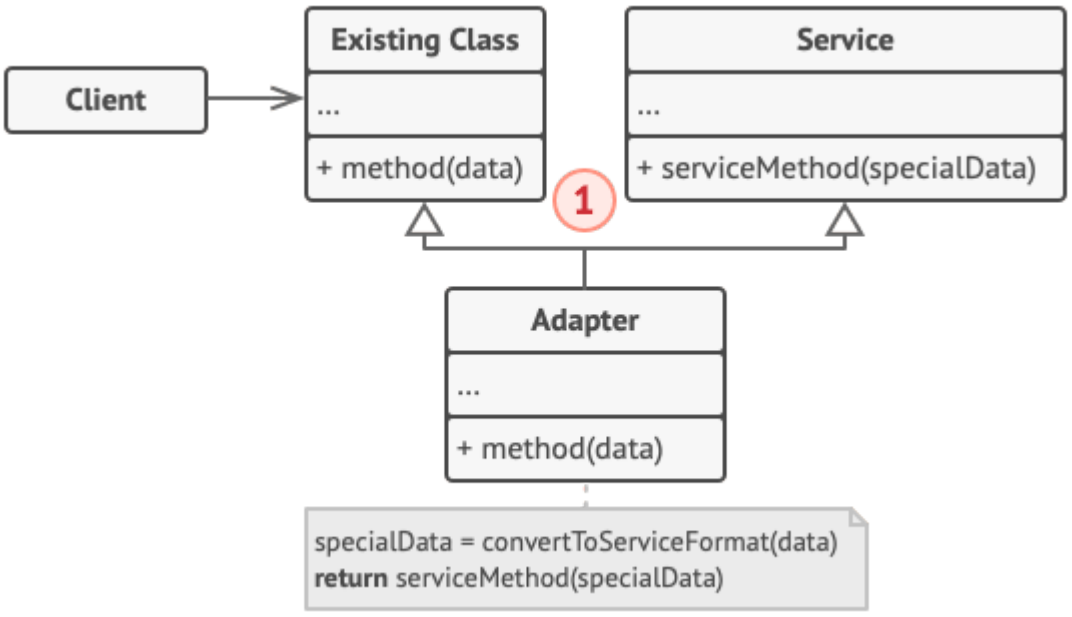
Object Adapter:

- Lets a single Adapter work with many Adaptees, including all of its subclasses (if any).
- Makes it harder to override Adaptee behavior. It requires subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Issues to consider when using the Adapter pattern:

- How much the Adapter should do. It should do only as much as it has to in order to adapt. If the Target and Adaptee are similar, then the adapter has to delegate the requests from the Target to the Adaptee. If they are not similar, then the adapter might have to convert the data structures between them and implement the operations required by the Target but not implemented by the Adaptee.
- Using two-way adapters to provide transparency. A potential problem with adapters is that they aren't transparent to all clients. An adapted object no longer conforms to the Adaptee interface, so it can't be used as is wherever an Adaptee object can. Two-way adapters can provide such transparency. Specifically, they're useful when two different clients need to view an object differently.

Feature	Class Adapter	Object Adapter
Composition vs Inheritance	Uses inheritance to adapt Adaptee	Uses composition (delegation) to adapt Adaptee
Subclassing Adaptee	Must subclass both Target and Adaptee	Does not require subclassing Adaptee
Adapting subclasses	Cannot adapt subclasses of Adaptee	Can adapt both Adaptee and its subclasses
Overriding Adaptee's behavior	Adapter can override some of Adaptee's behavior	Harder to override Adaptee's behavior
Code complexity	Requires less code for delegation	Requires more code for delegation
Interface adaptation	Can only adapt specific Adaptee class	Can adapt any Adaptee and its subclasses



Class adapter

Facade pattern

Facade patterns are beneficial when:

1. You need to provide a simplified interface to a complex subsystem. A facade can offer a simple default view of the subsystem that is sufficient for most clients.
2. You want to decouple the subsystem from clients and other subsystems, promoting subsystem independence and portability.
3. You need an entry point to each subsystem level.
4. You want to simplify the dependencies between subsystems by making them communicate solely through their facades.

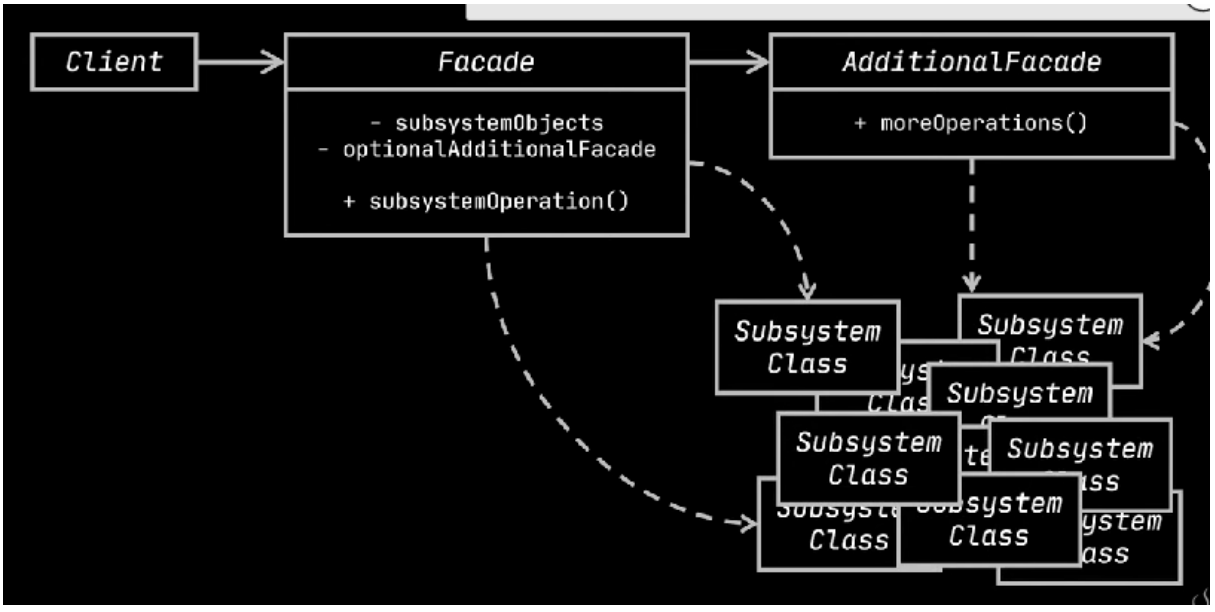
The advantages of using the Facade pattern include:

- **Simplification**: It shields clients from subsystem components, reducing the number of objects that clients need to deal with and making the subsystem easier to use.
- **Coupling reduction**: It promotes weak coupling between the subsystem and its clients, which can eliminate complex or circular dependencies. This is particularly important when the client and the subsystem are implemented independently.

- **Compilation dependencies**: Reducing compilation dependencies with facades can limit the recompilation needed for a small change in an important subsystem.

Some known uses of the Facade pattern include:

- **ET++ application framework**: It has built-in browsing tools for inspecting objects at runtime. These browsing tools are implemented in a separate subsystem, including a facade class called "ProgrammingEnvironment," which defines operations such as InspectObject and InspectClass for accessing the browsers.
- **Choices operating system**: It uses facades to compose many frameworks into one. Choices has key abstractions like processes, storage, and address spaces. For each of these abstractions, there is a corresponding subsystem, implemented as a framework, that supports porting Choices to various hardware platforms.



To implement the Facade pattern:

1. **Reduce client-subsystem coupling**:
    - Create a "Facade Abstract Class" and concrete subclasses for implementing the subsystem.
    - The client class can communicate with the subsystem through the "Abstract Facade Class."

- Alternatively, you can configure a facade object with different subsystem objects.

2. **Public versus Private Interfaces**:
   - Classes and subsystems are similar in the sense that both encapsulate something.
   - Both have private and public interfaces.
   - The public interface consists of classes accessible by all clients, whereas the private interface is only for subsystem extenders.
   - Facade is part of the public interface.

Here's how you can implement these steps:

```
// Facade Abstract Class
public abstract class AbstractFacade {
    // Method to provide a unified interface to the subsyst
em
    public abstract void operation();
}

// Concrete Facade Class
public class ConcreteFacade extends AbstractFacade {
    private Subsystem1 subsystem1;
    private Subsystem2 subsystem2;

    public ConcreteFacade() {
        this.subsystem1 = new Subsystem1();
        this.subsystem2 = new Subsystem2();
    }

    // Unified interface method that delegates to the subsy
stems
    @Override
    public void operation() {
        subsystem1.operation1();
        subsystem2.operation2();
    }
}
```

```
// Subsystem 1
class Subsystem1 {
    public void operation1() {
        System.out.println("Subsystem 1 operation");
    }
}

// Subsystem 2
class Subsystem2 {
    public void operation2() {
        System.out.println("Subsystem 2 operation");
    }
}

// Client class
public class Client {
    public static void main(String[] args) {
        AbstractFacade facade = new ConcreteFacade();
        facade.operation();
    }
}
```

In this implementation:

- `AbstractFacade` provides the unified interface.
- `ConcreteFacade` implements this interface and internally communicates with the subsystems.
- `Subsystem1` and `Subsystem2` are the classes representing the subsystems.
- The client class communicates with the subsystems through the facade, reducing coupling.

Proxy

Proxy design pattern - Intent

The Proxy pattern provides a surrogate or placeholder for another object to control access to it.

Intent:

- The Proxy pattern is heavily used to implement lazy loading related use cases where we do not want to create the full object until it is actually needed.
- Using the Proxy pattern, a class represents the functionality of another class.

In the Proxy Design Pattern, a client does not directly talk to the original object; it delegates calls to the proxy object, which calls the methods of the original object. Moreover, the important point is that the client does not know about the proxy. The proxy acts as an original object for the client.

There are three main variations of the Proxy Pattern:

1. A remote proxy provides a local representative for an object in a different address space.
2. A virtual proxy creates expensive objects on demand.
3. A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

Design participants:

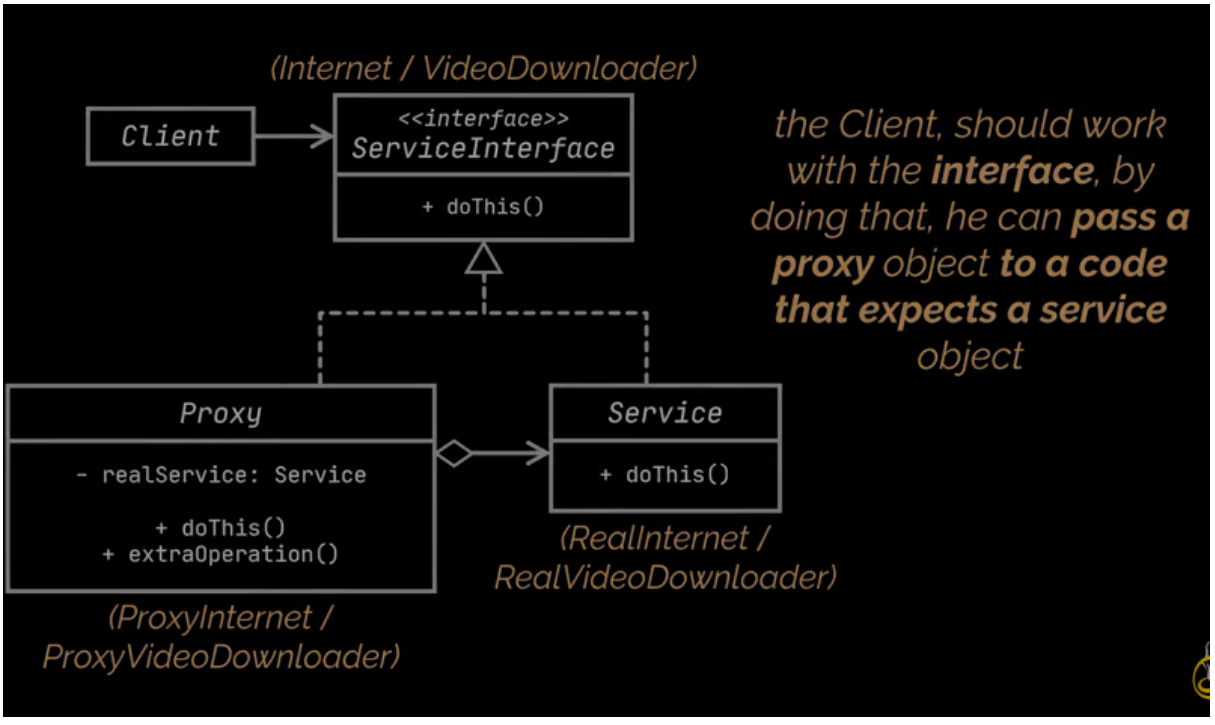
- **Subject:** An interface that exposes the functionality available to be used by the clients.
- **Real Subject:** A class implementing Subject. It is a concrete implementation that needs to be hidden behind a proxy.
- **Proxy:** Hides the real object by extending it, and clients communicate with the real object via this proxy object. Usually, frameworks create this proxy object when clients request the real object.

```java
public interface Internet {
    void connectTo(String host);
}

public class RealInternet implements Internet {
    @Override
    public void connectTo(String host) {
        System.out.println("Opened connection to " + host);
    }
}
```

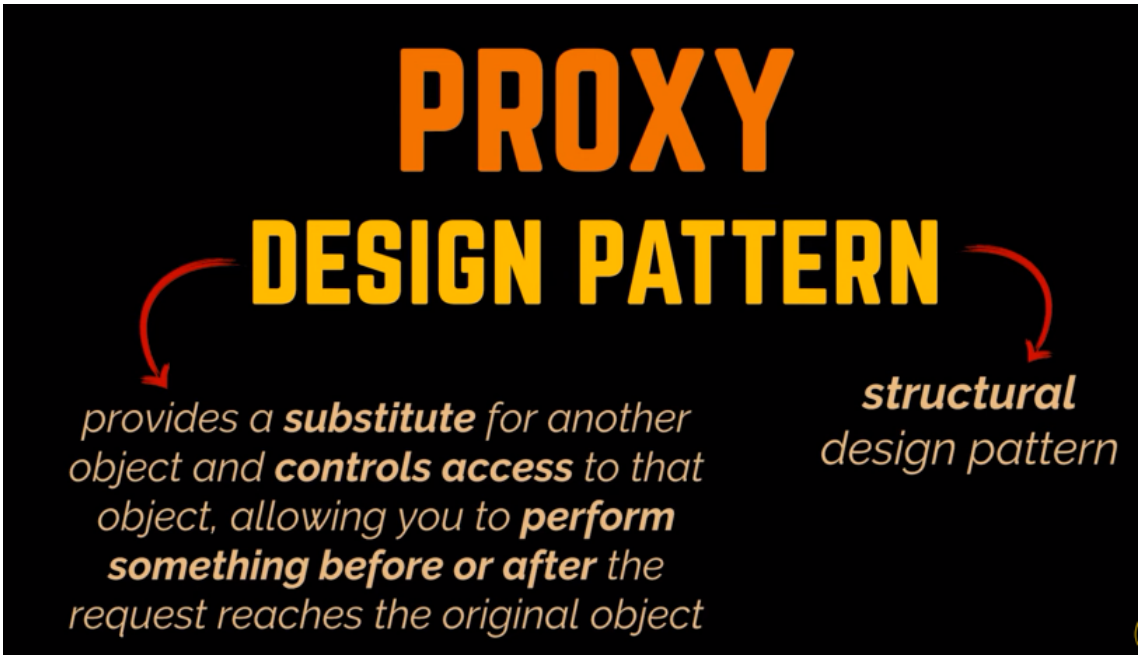
```java
public class ProxyInternet implements Internet {
    private static final List<String> bannedSites;
    private final Internet internet = new RealInternet();

    static {
        bannedSites = new ArrayList<>();
        bannedSites.add("banned.com");
    }

    @Override
    public void connectTo(String host) {
        if (bannedSites.contains(host)) {
            System.out.println("Access Denied!");
            return;
        }
        internet.connectTo(host);
    }
}
```

*now, the user who wants to benefit from the banned-websites functionality can use the proxy **without affecting other users***

```java
public static void main(String[] args) {
    Internet internet = new ProxyInternet();
    internet.connectTo("google.com");
    internet.connectTo("banned.com");
}
```



*the Client, should work with the **interface**, by doing that, he can **pass a proxy** object **to a code that expects a service** object*



When to use proxy design pattern - Applicability

The Proxy Pattern can be used in the following scenarios:

1. **Cache Proxy (Caching request results):**
    - Improves responsiveness by caching relevant data (that does not change frequently or is expensive to create) at the Proxy object level.
    - When a client sends a request, the Proxy checks if the requested data is present. If the data is found, it is returned to the client without sending the request to the Real Object.
2. **Remote Proxy (Local execution of a remote service):**
    - Provides a proxy object at a local level, representing an object present at another location.
3. **Protection Proxy (Access control):**
    - Provides security for Real Subject at the proxy level.
4. **Virtual Proxy (Lazy initialization):**
    - Used when the Real Object is a complex object or expensive to create.
    - Suitable for scenarios where a heavyweight service object wastes system resources by always being up, even though it is only needed from time to time.

Advantages of Proxy Design Pattern:

- The proxy works even if the service object isn't ready or is not available.
- Supports the Open/Closed Principle - new proxies can be introduced without changing the service or clients.

Disadvantages of Proxy Design Pattern:

- Introduces an additional layer between the client and the Real Subject, contributing to code complexity.
- The response from the service might get delayed.
- Additional request forwarding is introduced between the client and the Real Subject.

Problem Statement:

The library provides us with a video downloading class. However, it's very inefficient. If the client application requests the same video multiple times, the library just downloads it over and over, instead of caching and reusing the first downloaded file.

Solution using Proxy Design Pattern:

The proxy class implements the same interface as the original downloader and delegates all the work to it. However, it keeps track of the downloaded files and returns the cached result when the app requests the same video multiple times.

Applicability of Proxy Design Pattern:

- **Cache Proxy (Caching request results):**
    - Improves responsiveness by caching relevant data (that does not change frequently or is expensive to create) at the Proxy object level.
    - When a client sends a request, the Proxy checks if the requested data is present. If the data is found, it is returned to the client without sending the request to the Real Object.

Advantages of Proxy Design Pattern:

- The proxy works even if the service object isn't ready or is not available.
- Supports the Open/Closed Principle - new proxies can be introduced without changing the service or clients.

Disadvantages of Proxy Design Pattern:

- Introduces an additional layer between the client and the Real Subject, contributing to code complexity.
- The response from the service might get delayed.
- Additional request forwarding is introduced between the client and the Real Subject.

```
import java.util.HashMap;
import java.util.Map;

// Subject interface
interface VideoDownloader {
    void download(String videoId);
}

// Real Subject
class RealVideoDownloader implements VideoDownloader {
    @Override
    public void download(String videoId) {
        System.out.println("Downloading video " + videoId);
        // Simulate downloading the video
    }
}

// Proxy
class ProxyVideoDownloader implements VideoDownloader {
    private RealVideoDownloader downloader;
    private Map<String, Boolean> cache;

    public ProxyVideoDownloader() {
        this.downloader = new RealVideoDownloader();
        this.cache = new HashMap<>();
    }

    @Override
    public void download(String videoId) {
        if (!cache.containsKey(videoId) || !cache.get(video
Id)) {
```

```
            downloader.download(videoId);
            cache.put(videoId, true);
        } else {
            System.out.println("Video " + videoId + " alrea
dy downloaded. Retrieving from cache.");
        }
    }
}

// Client
public class Client {
    public static void main(String[] args) {
        VideoDownloader downloader = new ProxyVideoDownload
er();

        // Download video for the first time
        downloader.download("video1");

        // Try to download the same video again
        downloader.download("video1");

        // Download a different video
        downloader.download("video2");
    }
}
```

Output:

```
Downloading video video1
Video video1 already downloaded. Retrieving from cache.
Downloading video video2
```

In this example, `RealVideoDownloader` represents the actual downloader class, `ProxyVideoDownloader` acts as a proxy, and `Client` demonstrates the usage of the proxy.

Flyweight

FLYWEIGHT DESIGN PATTERN

*lets you **fit more objects** into the available RAM by **sharing common parts of state between multiple objects**, instead of storing all of the data in each object individually*

***structural** design pattern*

```java
@Data
public class Book {
    private final String name;
    private final double price;
    private final BookType type;
}
```

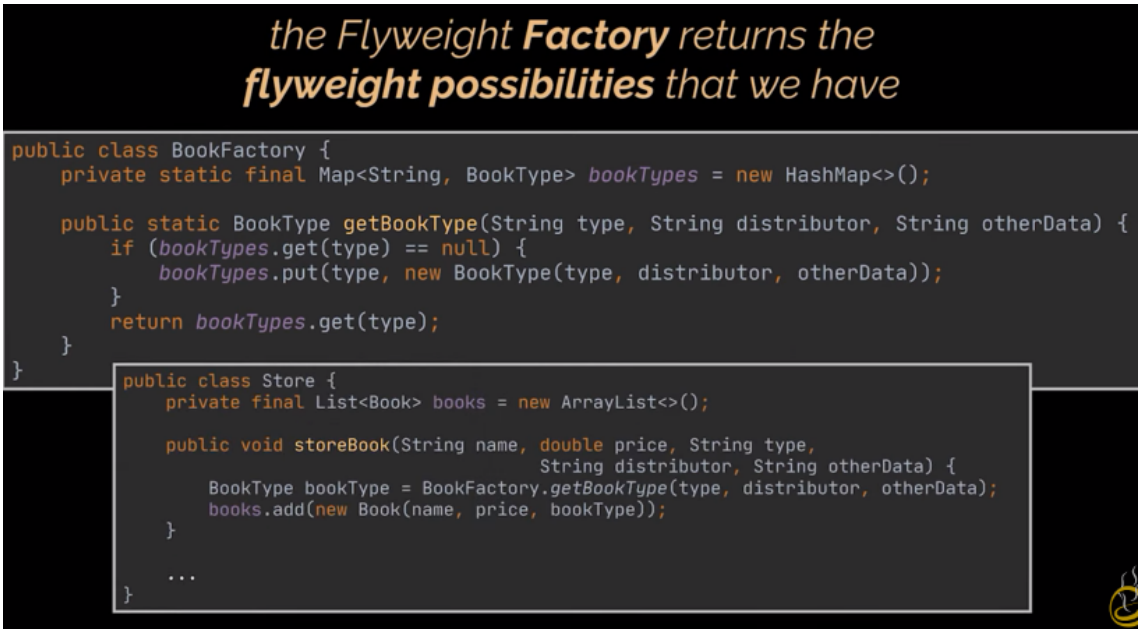
```java
@Getter
@AllArgsConstructor
public class BookType {
    private final String type;
    private final String distributor;
    private final String otherData;
}
```

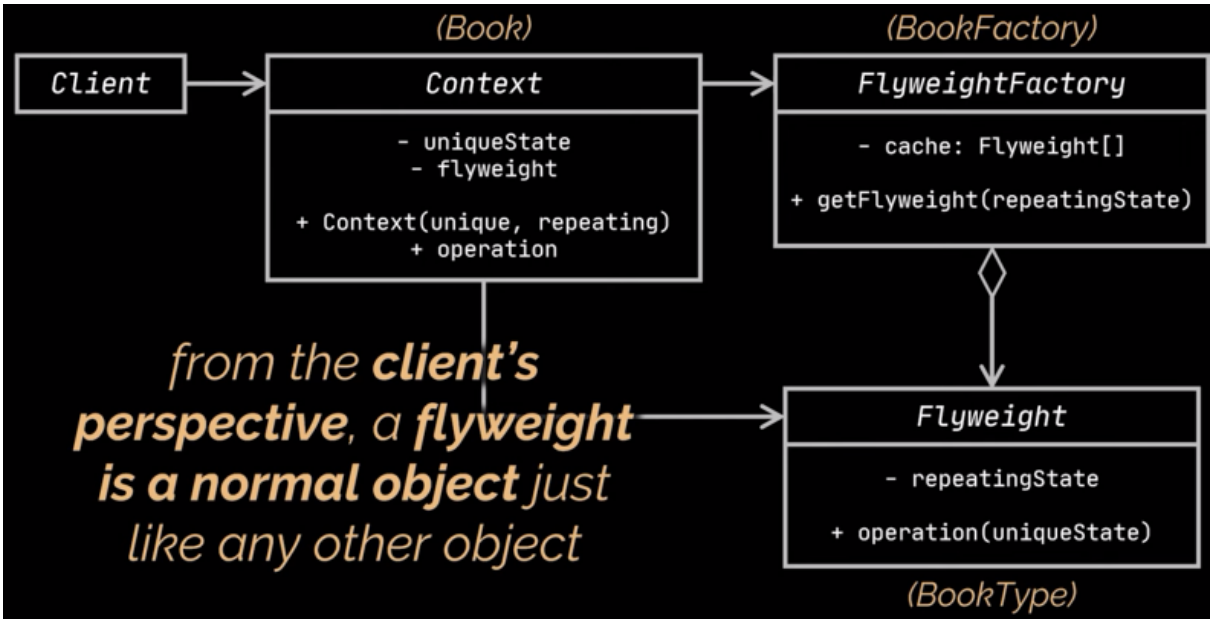
flyweight class

*flyweights are **immutable***

*it stores the **intrinsic** state of the object*

*this state is **invariant**, **context-independent**, **shareable** and **never altered** at runtime*





Flyweight Pattern - Intent:

The flyweight pattern is for sharing objects, where each instance does not contain its own state but stores it externally. This allows efficient sharing of objects to save space when there are many instances but few different types.

Motivation:

- Placing the same information in many different places leads to:
  1. Mistakes in copying the information.
  2. Difficulty in maintaining the document if the data changes.

3. Large documents if the same information is repeated over and over again.

- Issues:
  1. RAM overhead associated with each instance.
  2. CPU overhead associated with memory management.

Solution:

- Create Flyweight Objects for Intrinsic state:
  - **Intrinsic State:** Information independent of the object's context, shareable (e.g., state might include name, postal abbreviation, time zone, etc.). This is included in the flyweight and stored only once instead of n times for n objects. **Common Values**
  - **Extrinsic State:** Information dependent on the object's context, unshareable, stateless, having no stored values, but values that can be calculated on the spot (e.g., state might need access to the region). This is excluded from the Flyweight. **Runtime Values**

```
import java.util.HashMap;
import java.util.Map;

// Flyweight interface
interface Flyweight {
    void operation(String extrinsicState);
}

// Concrete Flyweight
class ConcreteFlyweight implements Flyweight {
    private String intrinsicState;

    public ConcreteFlyweight(String intrinsicState) {
        this.intrinsicState = intrinsicState;
    }

    @Override
    public void operation(String extrinsicState) {
        System.out.println("ConcreteFlyweight: Intrinsic st
```

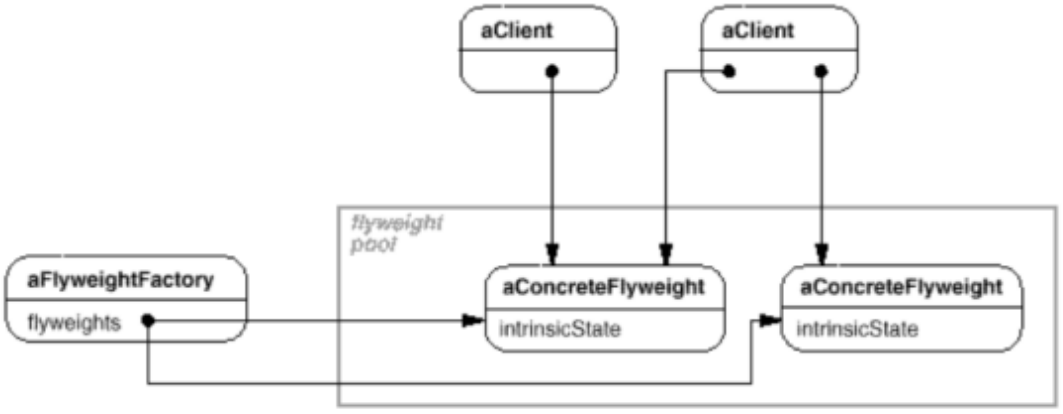
```
ate = " + intrinsicState + ", Extrinsic state = " + extrins
icState);
    }
}

// Flyweight Factory
class FlyweightFactory {
    private Map<String, Flyweight> flyweights = new HashMap
<>();

    public Flyweight getFlyweight(String key) {
        if (!flyweights.containsKey(key)) {
            flyweights.put(key, new ConcreteFlyweight(ke
y));
        }
        return flyweights.get(key);
    }
}

// Client
public class Client {
    public static void main(String[] args) {
        FlyweightFactory factory = new FlyweightFactory();

        // Using flyweight objects with intrinsic state "A"
        Flyweight flyweight1 = factory.getFlyweight("A");
        flyweight1.operation("1");

        Flyweight flyweight2 = factory.getFlyweight("A");
        flyweight2.operation("2");

        // Using flyweight objects with intrinsic state "B"
        Flyweight flyweight3 = factory.getFlyweight("B");
        flyweight3.operation("1");

        Flyweight flyweight4 = factory.getFlyweight("B");
        flyweight4.operation("2");
```

```
    }
}
```

Output:

```
ConcreteFlyweight: Intrinsic state = A, Extrinsic state = 1
ConcreteFlyweight: Intrinsic state = A, Extrinsic state = 2
ConcreteFlyweight: Intrinsic state = B, Extrinsic state = 1
ConcreteFlyweight: Intrinsic state = B, Extrinsic state = 2
```

In this example, `ConcreteFlyweight` represents the flyweight objects, `FlyweightFactory` creates and manages flyweight objects, and `Client` demonstrates the usage of flyweight objects.



Flyweight Participants:

- **Flyweight:**
  - Declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight (Character):**
  - Implements the Flyweight interface and adds storage for intrinsic state, if any.
  - Must be sharable, and any state it stores must be intrinsic, independent of the ConcreteFlyweight object's context.

- **UnsharedConcreteFlyweight (Row, Column):**
  - Not all Flyweight subclasses need to be shared.
  - The Flyweight interface enables sharing but doesn't enforce it.
  - It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
- **FlyweightFactory:**
  - Creates and manages flyweight objects.
  - Ensures that flyweights are shared properly.
  - When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one if none exists.
- **Client:**
  - Maintains a reference to flyweight(s).
  - Computes or stores the extrinsic state of flyweight(s).

Applicability:

- An application uses a large number of objects.
- Storage costs are high due to the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Consequences:

- **Removing extrinsic state:** The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects. Removing extrinsic state won't help reduce storage costs if there are as many different kinds of extrinsic state as there are objects before sharing.
- **Managing shared objects:** Because objects are shared, clients shouldn't instantiate them directly. FlyweightFactory lets clients locate a particular

flyweight. FlyweightFactory objects often use an associative store to let clients look up flyweights of interest.

Examples:

- **Pen with Refill:** A pen can exist with or without a refill, and a refill can be of any color. In this scenario, a pen can be a flyweight object with a refill as an extrinsic attribute. All other attributes, such as the pen body and pointer, can be intrinsic attributes that are common to all pens. Each pen will be distinguished by its refill color only.
- **java.lang.String Constants:** All strings are stored in the string pool, and if we need a string with certain content, the runtime returns the reference to an already existing string constant from the pool, if available.
- **Browsers:** Browsers use images in multiple places on a webpage. Browsers load the image only once and reuse it from the cache for other instances. The URL of the image is an intrinsic attribute because it's fixed and shareable, while the image's position coordinates, height, and width are extrinsic attributes that vary according to the place (context) where they have to be rendered.

Flyweight and immutability

Since the same flyweight object can be used in different contexts, you have to make sure that its state can’t be modified. A flyweight should initialize its state just once, via constructor parameters. It shouldn’t expose any setters or public fields to other objects.

Anti-Patterns

What is an Antipattern?

- An AntiPattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences.
- The AntiPattern may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type of problem, or having applied a perfectly good pattern in the wrong context.

Antipatterns Vs. Design Patterns

- **Design Pattern:** A general repeatable solution to a commonly occurring problem.
- **Antipattern:** A solution recognized as a poor way to solve the problem, often leading to negative consequences.

Why study Antipatterns?

- Antipatterns provide easily identifiable templates for common problems, as well as a path of action to rectify these problems.
- They offer real-world experience in recognizing recurring problems in the software industry, along with detailed remedies for the most common ones.
- Antipatterns provide a common vocabulary for identifying problems and discussing solutions.
- They offer stress relief in the form of shared misery.
- They ensure common problems are not continually repeated within an organization.

Viewpoints

AntiPatterns are categorized from three major viewpoints:

- **The software developer**
- **The software architect**
- **The software manager**

Principal AntiPattern Viewpoints:

- **Development AntiPatterns:** Describe situations encountered by the programmer when solving programming problems.
- **Architectural AntiPatterns:** Focus on common problems in system structure, their consequences, and solutions. Many of the most serious unresolved problems in software systems occur from this perspective.
- **Management AntiPatterns:** Describe common problems and solutions due to the software organization. They affect people in all software roles, and their solutions directly affect the technical success of the project.

Reference Model

A reference model for terminology common to all three viewpoints. The reference model is based upon three topics that introduce the key concepts of AntiPatterns:

Root Causes:

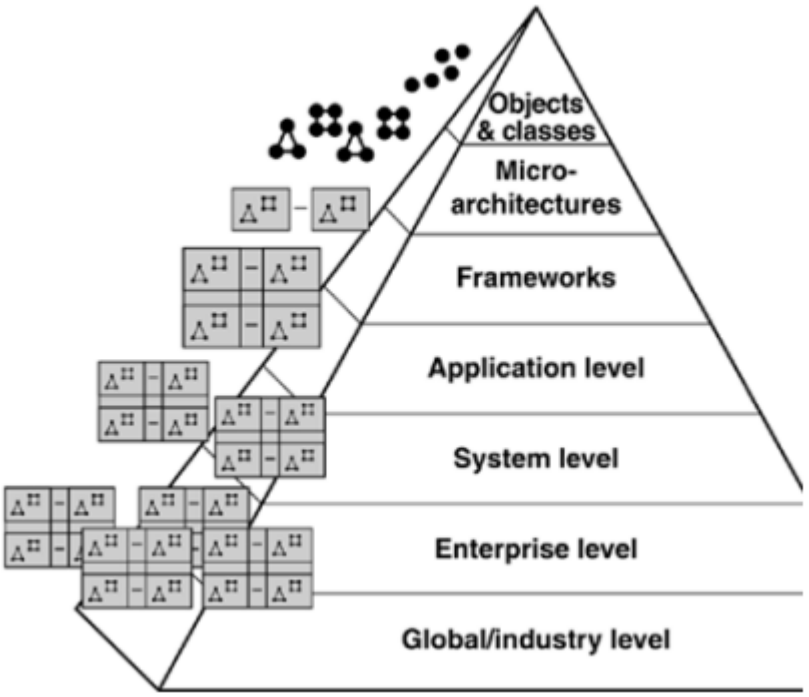
- Root causes are mistakes in software development that result in failed projects, cost overruns, schedule slips, and unfulfilled business needs.
- The root causes are based upon the “seven deadly sins,” a popular analogy that has been used successfully to identify ineffective practices.

1. **Haste:** Tight deadlines often lead to neglecting important activities.
2. **Apathy:** The attitude of not caring about solving known problems.
3. **Narrow-Mindedness:** Refusal of developers to learn proven solutions.
4. **Sloth:** Adaptation of the most simple “solution”.
5. **Avarice:** Greed in creating a system can result in very complex and difficult-to-maintain software.
6. **Ignorance:** The lack of motivation to understand things.
7. **Pride:** The failure to reuse existing software packages because they were not invented by a specific company.

Primal Forces:

Forces are concerns or issues that exist within a decision-making context. In a design solution, forces that are successfully addressed (or resolved) lead to benefits, and forces that are unresolved lead to consequences. The primal forces include:

1. **Management of Functionality:** Meeting the requirements.
2. **Management of Performance:** Meeting required speed and operation.
3. **Management of Complexity:** Defining abstractions.
4. **Management of Change:** Controlling the evolution of software.
5. **Management of IT Resources:** Managing people and IT artifacts.
6. **Management of Technology Transfer:** Controlling technology evolution.



Level Model

- The global level contains the design issues that are globally applicable across all systems. This level is concerned with coordination across all organizations, which participate in cross-organizational communications and information sharing.
- The enterprise level is focused upon coordination and communication across a single organization. The organization can be distributed across many locations and heterogeneous hardware and software systems.
- The system level deals with communications and coordination across applications and sets of applications.
- The application level is focused upon the organization of applications developed to meet a set of user requirements. The macro–component levels are focused on the organization and development of application frameworks.
- The micro–component level is centered on the development of software components that solve recurring software problems. Each solution is relatively self–contained and often solves just part of an even larger problem.

- The object level is concerned with the development of reusable objects and classes. The object level is more concerned with code reuse than design reuse. Each of the levels is discussed in detail along with an overview of the patterns documented at each level.

Project Management AntiPattern – Analysis Paralysis

Symptoms and Consequences:

- Multiple project restarts and significant model rework due to personnel changes or changes in project direction.
- Design and implementation issues are continually reintroduced in the analysis phase.
- Cost of analysis exceeds expectations without a predictable endpoint.
- The analysis phase no longer involves user interaction, and much of the analysis performed is speculative.
- The complexity of the analysis models results in intricate implementations, making the system difficult to develop, document, and test.
- Design and implementation decisions, such as those used in the Gang of Four design patterns, are made in the analysis phase.

Typical Causes:

- The management process assumes a waterfall progression of phases, while virtually all systems are built incrementally, even if not acknowledged in the formal process.
- Management has more confidence in their ability to analyze and decompose the problem than to design and implement.
- Management insists on completing all analysis before the design phase begins.
- Goals in the analysis phase are not well-defined.
- Planning or leadership lapses when moving past the analysis phase.

- Management is unwilling to make firm decisions about when parts of the domain are sufficiently described.
- The project vision and focus on the goal/deliverable to the customer are diffused. Analysis goes beyond providing meaningful value.

Software Architecture Antipatterns - Vendor Lock-In

Symptoms and Consequences:

- Commercial product upgrades drive the application software maintenance cycle.
- Promised product features are delayed or never delivered, subsequently causing a failure to deliver application updates.
- The product varies significantly from the advertised open systems standard.
- If a product upgrade is missed entirely, a product repurchase and reintegration are often necessary.
- The product varies from published open system standards because there is no effective conformance process for the standard.
- The product is selected based entirely upon marketing and sales information, and not upon more detailed technical inspection.
- There is no technical approach for isolating application software from direct dependency upon the product.
- Application programming requires in-depth product knowledge.
- The complexity and generality of the product technology greatly exceed that of the application needs; direct dependence upon the product results in a failure to manage the complexity of the application system architecture.

Typical Causes:

- The product is selected based entirely upon marketing and sales information, without more detailed technical inspection.
- There is no effective conformance process for the open system standard.
- Application programming requires in-depth product knowledge.

- The complexity and generality of the product technology greatly exceed that of the application needs, leading to a failure to manage the complexity of the application system architecture.

Refactored Solution – Isolation Layer:
An isolation layer separates software packages and technology. This solution is applicable when one or more of the following conditions apply:

- Isolation of application software from lower-level infrastructure.
- Changes to the underlying infrastructure are anticipated within the life cycle of the affected software.
- A more convenient programming interface is useful or necessary.
- There is a need for consistent handling of the infrastructure across many systems.
- Multiple infrastructures must be supported, either during the life cycle or concurrently.

Sure, let's break it down into simpler terms:

Symptoms and Consequences:

- **Commercial product upgrades drive the application software maintenance cycle:** Imagine you have a software application that relies on another product or technology. Whenever this other product gets updated, you are forced to update your software too, leading to a cycle of constant maintenance.
- **Promised product features are delayed or never delivered:** You might expect new features or improvements in the product you're relying on, but they often get delayed or never delivered, leaving your software stuck without the improvements you were expecting.
- **The product varies significantly from the advertised open systems standard:** The product you're using doesn't behave the way you expected it to, based on the standards it claims to follow.
- **If a product upgrade is missed entirely, a product repurchase and reintegration are often necessary:** If you don't update your software to match the new version of the product you're relying on, you might need to buy the product again and rework your software to make it compatible.

- **Application programming requires in-depth product knowledge:** Building your software requires a deep understanding of the product you're using, which adds complexity to your development process.
- **The complexity and generality of the product technology greatly exceed that of the application needs:** The product you're using is much more complex than what your application actually needs, making it hard to manage and maintain your software.

Typical Causes:

- **The product is selected based entirely upon marketing and sales information:** You choose a product based on how it's advertised rather than how well it suits your needs.
- **There is no effective conformance process for the open system standard:** The product doesn't follow the industry standards properly.
- **Application programming requires in-depth product knowledge:** Building your software requires a deep understanding of the product you're using, which adds complexity to your development process.
- **The complexity and generality of the product technology greatly exceed that of the application needs:** The product you're using is much more complex than what your application actually needs, making it hard to manage and maintain your software.

Refactored Solution – Isolation Layer:
To solve these problems, you can create an isolation layer between your software and the product you're relying on. This means separating your software from the inner workings of the product. It's like building a buffer between your software and the product, so that even if the product changes, your software can continue to work without needing constant updates.

Software Development AntiPattern – The Blob

Background:
The Blob AntiPattern is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data. This AntiPattern is characterized by a class diagram composed of a single complex controller class surrounded by simple data classes. The key problem here is that the majority of the responsibilities are allocated to a single class.

Symptoms and Consequences:

- Single class with a large number of attributes, operations, or both. A class with 60 or more attributes and operations usually indicates the presence of the Blob.
- A disparate collection of unrelated attributes and operations encapsulated in a single class. An overall lack of cohesiveness of the attributes and operations is typical of the Blob.
- The Blob Class is typically too complex for reuse and testing. It may be inefficient or introduce excessive complexity to reuse the Blob for subsets of its functionality.
- The Blob Class may be expensive to load into memory, using excessive resources, even for simple operations.

Typical Causes:

- Lack of an object-oriented architecture.
- Lack of any architecture.
- Lack of architecture enforcement.
- Too limited intervention.
- Specified disaster.

Refactored Solution

Step 1:

- Identify or categorize related attributes and operations according to contracts. These contracts should be cohesive in that they all directly relate to a common focus, behavior, or function within the overall system.

Step 2:

- Look for "natural homes" for these contract-based collections of functionality and then migrate them there. In this example, we gather operations related to catalogs and migrate them from the LIBRARY class and move them to the CATALOG class.

Step 3:

- Remove all "far-coupled," or redundant, indirect associations. In the example, the ITEM class is initially far-coupled to the LIBRARY class in that each item really belongs to a CATALOG, which in turn belongs to a LIBRARY.

- Where appropriate, migrate associates to derived classes to a common base class. In the example, once the far-coupling has been removed between the LIBRARY and ITEM classes, we need to migrate ITEMs to CATALOGs.
- Remove all transient associations, replacing them as appropriate with type specifiers to attributes and operations arguments.

Sure, here's an example of the Blob antipattern:

```
// Blob class
public class Library {
    private List<Catalog> catalogs;
    private List<Item> items;
    private List<Customer> customers;
    private List<Transaction> transactions;
    private List<Staff> staff;

    // Methods
    public void addItem(Item item) {
        // Add item to library
    }

    public void removeItem(Item item) {
        // Remove item from library
    }

    public void addCatalog(Catalog catalog) {
        // Add catalog to library
    }

    public void removeCatalog(Catalog catalog) {
        // Remove catalog from library
    }

    // Similarly, more methods to manage customers, staff,
and transactions
}

public class Catalog {
```

```
    private List<Item> items;

    // Methods
    public void addItem(Item item) {
        // Add item to catalog
    }

    public void removeItem(Item item) {
        // Remove item from catalog
    }
}

public class Item {
    private String name;
    private String author;
    private int year;
    private double price;

    // Methods
    // Getters and setters for attributes
}

// Other classes like Customer, Transaction, and Staff
```

In this example, the `Library` class contains a large number of attributes and methods, making it the Blob. It holds most of the responsibilities, while other classes like `Catalog`, `Item`, `Customer`, `Transaction`, and `Staff` primarily hold data or execute simple processes.

The `Library` class violates the Single Responsibility Principle and lacks cohesion, making it difficult to maintain, reuse, and test. To refactor this antipattern, we need to distribute responsibilities more uniformly among classes.