The Business Software Development Series

GoF Design Patterns Distilled

Dominik Cebula

Table of Contents

Chapter 1. Introduction

1.1. Overview

GoF (Gang of Four) Design Patterns are a collection of 23 design patterns introduced in the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. These patterns provide solutions for common problems that occur in software development. They are divided into three groups: Creational, Structural, and Behavioral. Although the original book provides examples in C++, GoF Design Patterns are applicable to any language that supports Object-Oriented Programming (OOP) paradigm.

In this book, I want to provide straight to a point, distilled explanation of GoF Design Patterns, including examples and my point of view on this topic.

1.2. How to learn design patterns

Learning the design patterns involves some back-and-forth process. Pattern introduction usually starts with some sort of an overview or a definition that tries to capture general information and problem that patterns solve. Challenge is that usually, this definition is hard to understand when you do not know the pattern, yet it makes perfect sense after you understand the pattern. Because of that, I recommend learning design patterns in the following way:

1. Read the overview section.
2. Read the use cases section.
3. Read the general structure and UML diagram.
4. Read / understand / practice on example code.
5. Revisit the overview section.
6. Revisit the use cases section.
7. Revisit the general structure and UML diagram.
8. Implement your own example.
9. Practice during your daily job

1.3. Definition of a pattern

A pattern is a reusable solution to a problem frequently occurring in a particular context. Patterns are not specific solutions; instead, they provide guidelines for solving problems that can be adapted to different situations. A pattern describes a problem, its solution, and the context in which it applies.

1.4. Benefits of patterns

The use of design patterns in software development has many benefits. Patterns provide a proven solution to common problems, reducing the risk of errors and increasing the software's reliability. They also give the developers a common vocabulary, improving communication and reducing the time needed to share ideas. Patterns also improve the maintainability of software by making it easier to understand and modify.

1.5. Criticism of patterns

Despite the many benefits of design patterns, they are not without criticism. One criticism is that patterns can be overused, leading to complex and overly abstract code. Another complaint is that patterns can be misused, making code harder to understand and maintain.

1.6. Not everything has to be a pattern

When writing business software, always have the main goal in mind of writing business logic in a simple, clear, easy-to-understand, and maintainable manner. Design Patterns should be used if they fall naturally into the use case that you currently implement. Once you learn all design patterns, you should not force them. Instead, you should keep them as a tool ready to be utilized whenever you struggle with presenting a business problem, as a code in an easy-to-understand way. Design Patterns should inspire you to structure your code to present behaviors clearly. If at any moment you see that your code is getting over complex because of Design Pattern, you should verify if the correct pattern was chosen for the problem that you are trying to solve. In some cases relying only on programming language capabilities might be good enough.

1.7. Relation to other principles

Design Patterns are important, and they can improve the way that you write the code. However, it is important to take into account that the following principles take prejudice over design patterns:

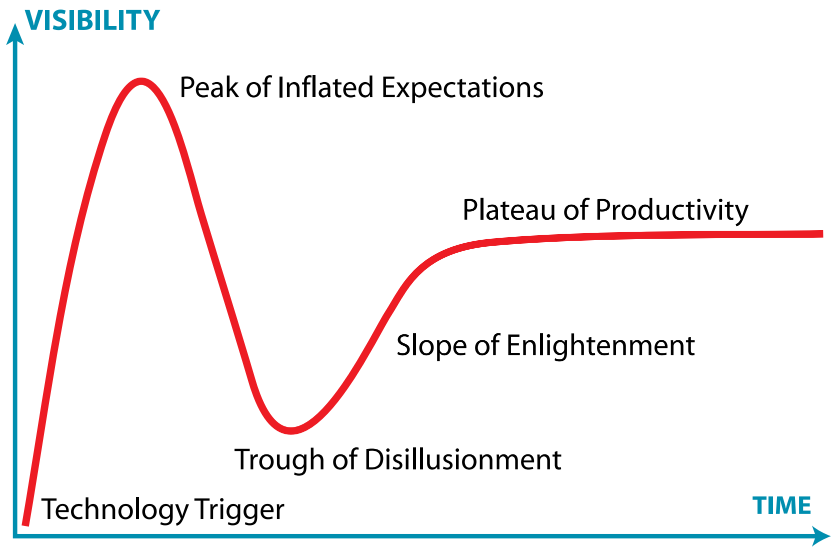
- Clean Code
- KISS – Keep it simple
- DRY – Do not repeat yourself
- YAGNI – You aren't gonna need it
- Design for high cohesion, low coupling
- SOLID
  - SRP – Single Responsibility Principle
  - OCP – Open-Close Principle
  - LSP – Liskov Substitiude Principle

    - ISP – Interface Segregation Principle
    - DIP – Dependency Inversion Principle
- Principles of Component Cohesion
    - CCP – Common Closure Principle
    - CRP – Common Reuse Principle
    - REP – Reuse/Release Equivalence Principle
- Principles of Component Coupling
    - ADP – Acyclic Dependencies Principle
    - SDP – Stable-Dependency Principle
    - SAP – Stable-Abstractions Principle

Covering all of the above principles is beyond the scope of this book. However, I am mentioning the above because, in some cases, the above rules allow you to identify that misused pattern. For example, if you apply some pattern and observe code duplicates because of the used pattern, this clearly indicates that the pattern is misused.

1.8. Hype and patterns adoption

Like any concept or technology, I believe the design patterns also follow Gartner hype cycle:



Usually, I see the following phases when learning and adopting the design patterns:

1. Problem phase – Code is hard to read and understand, code is under-engineered.
2. Solution discovery phase – GoF Design Patterns are discovered.
3. Hype phase – Code is over-engineered. Everything becomes a pattern. We no longer have a simple constructor, instead factory or builder is used. We no longer have a simple loop, instead visitor, iterator or chain of responsibility is used.

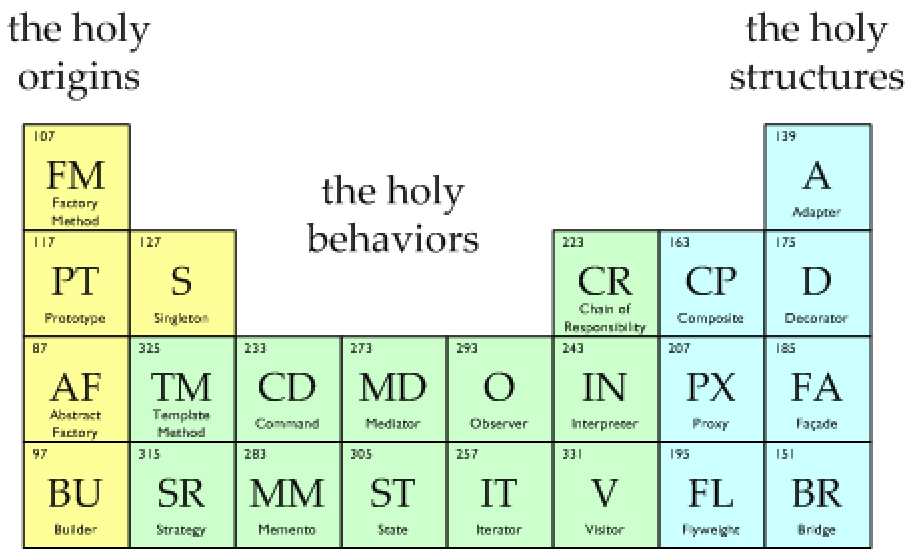
4. Doubt phase – GoF Design Patterns made the code even worse.
5. Enlightenment phase – let’s try to figure out why the design pattern did not help. Let’s identify when those should and should not be used.
6. Proper solution and adoption phase – GoF Design Patterns are used when they improve code structure and readability, in other cases, normal programming language constructs are used.

1.9. Design Patterns Types

GoF Design Patterns are divided into three groups:

1. Creational Design Patterns.
2. Structural Design Patterns.
3. Behavioral Design Patterns.

Vince Huston created an excellent graphic that captures all Design Patterns divided into groups:



1.9.1. Creational Design Patterns

Creational design patterns are patterns that deal with the creation of objects. These patterns provide ways to create objects in a manner that is flexible, efficient, and easy to understand.

1. Factory Method Pattern: allows you to customize a class behavior by overriding the method used to create an object with which the class will collaborate.
2. Abstract Factory Pattern: allows for the creation of families of related objects.
3. Builder Pattern: allows creating complex objects using the same construction process and provides the flexibility of choosing attribute values with which object should be created.
4. Singleton Pattern: ensures that a class is created only once and provides global access to the instance.

5. Prototype Pattern: allows cloning of existing objects, providing a way to create new objects with the same attributes as existing ones.

1.9.2. Structural Design Patterns

Structural design patterns are patterns that deal with the composition of classes and objects.

1. Facade Pattern: provides a simplified interface to a complex subsystem consisting of a set of classes.
2. Composite Pattern: allows to work with individual objects and a group of objects in the same way.
3. Decorator Pattern: attaches extra responsibilities to an object dynamically, providing a flexible alternative to subclassing for extending functionality.
4. Adapter Pattern: allows classes with incompatible interfaces to work together by converting the interface of a class into another interface that clients expect.
5. Proxy Pattern: wraps original object, allowing for another object to intercept calls to original object.
6. Bridge Pattern: solves the issue of exponential growth of classes that may occur when decomposing the problem being solved into objects.
7. Flyweight Pattern: memory consumption optimization technique, reduces memory usage by sharing the common and immutable state across many objects.

1.9.3. Behavioral Design Patterns

Behavioral design patterns are all about the communication between objects. These patterns provide ways to manage the interactions between objects in a manner that is flexible, and efficient.

1. Strategy Pattern: encapsulates a family of algorithms, each one of them is made interchangeable within a particular context.
2. Template Method Pattern: defines the skeleton of an algorithm in a base class, while some detailed steps are defined in subclasses.
3. Chain of Responsibility Pattern: links multiple objects into the processing pipeline, which is traversed to handle the request.
4. Visitor Pattern: allows separating algorithms from the objects on which they operate, groups similar operations that should be applied to different objects.
5. Observer Pattern: defines a subscription mechanism between objects, allows to observe object state, when object state changes, multiple dependencies are notified.
6. Command Pattern: encapsulates the entire request as a stand-alone object.
7. Interpreter Pattern: provides a way to define a language, its grammar, and syntax in an object-oriented way.
8. Iterator Pattern: provides a way to access the collection of elements sequentially without exposing underlying representation details.

9. Mediator Pattern: reduces the amount of dependencies between objects, by introducing a mediator object. Objects then use this mediator object to communicate with each other instead of communicating directly.
10. Memento Pattern: allows restoring an object to its previous state.
11. State Pattern: allows to implement Finite-State Machine using the Object Oriented Approach, object behavior changes when its internal state changes.

1.10. Summary

GoF Design Patterns provide a set of best practices and solutions for common problems that occur in software development. These patterns are widely used by software developers and have many benefits, including improved reliability, maintainability, and communication. However, they are not without criticism and must be used appropriately to avoid complexity and maintainability issues. The patterns are divided into three categories: creational, structural, and behavioral, each providing a set of solutions for specific types of problems.

1.11. References

1. Erich G, Richard H, Ralph J, John V. Introduction. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. October 31, 1994.
2. Design Patterns. Refactoring Guru. Accessed March 27, 2023. https://refactoring.guru/design-patterns
3. Design Patterns. Sourcemaking. Accessed March 27, 2023. https://sourcemaking.com/design_patterns
4. Design Patterns. Vince Huston Homepage. Accessed March 27, 2023. http://www.vincehuston.org/dp/
5. Design Patterns. Computer Science Department New Mexico State University. Accessed March 27, 2023. https://www.cs.nmsu.edu/~rth/cs/cs371/patterns.html
6. Eph Baum Blog. A Swift Dive into the Gang of Four Design Patterns. Accessed March 27, 2023. https://ephbaum.dev/a-swift-dive-into-the-gang-of-four-design-patterns/
7. Baeldung. Abstract Factory Pattern in Java. Accessed March 27, 2023. https://www.baeldung.com/java-abstract-factory-pattern
8. Medium. Singleton pattern in Python: A Beginner’s Guide. Accessed March 27, 2023. https://medium.com/@yeaske/singleton-pattern-in-python-a-beginners-guide-75e97ce75554
9. The Builder Pattern. Accessed March 27, 2023. https://python-automation.tistory.com/208
10. CSCI: Object-Oriented Analysis & Design – Lecture 8. Facade & Adapter. Accessed March 27, 2023. https://home.cs.colorado.edu/~kena/classes/5448/f12/lectures/08-facadeadapter.pdf
11. Medium. Understanding the Facade Design Pattern. Accessed March 27, 2023. https://medium.com/@adamszpilewicz/understanding-the-facade-design-pattern-with-go-9b05e848837b

12. DEV Community. Daniel Lee. Design Pattern: The Adapter & Facade Patterns – DEV Community. Accessed March 27, 2023. https://dev.to/blowideas/design-pattern-the-adapter-facade-patterns-2ipf
13. GeeksforGeeks. The Decorator Pattern | Set 2 (Introduction and Design). Accessed March 27, 2023. https://www.geeksforgeeks.org/the-decorator-pattern-set-2-introduction-and-design/
14. Wikipedia. Gartner hype cycle. Accessed March 27, 2023. https://en.wikipedia.org/wiki/Gartner_hype_cycle

Chapter 2. Behavioral Patterns

2.1. Strategy

Not available in the preview.

2.2. Template Method

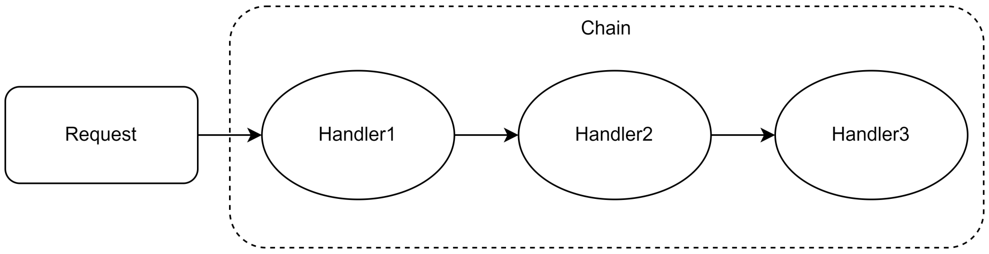
Not available in the preview.

2.3. Chain of Responsibility

2.3.1. Overview

The Chain of Responsibility Design Pattern is a behavioral pattern that allows you to chain together a series of objects to handle a request in a flexible and extensible way. This pattern is useful when you have a series of objects that can handle a request but don't know which one is capable of handling it. By using the Chain of Responsibility Design Pattern, you can pass the request through the chain of objects until one of them is able to handle it. This pattern is a good fit when the problem that is being solved requires flexibility of each element in the chain having autonomy in deciding when a request should stop processing and when this condition is different for each element of the chain.

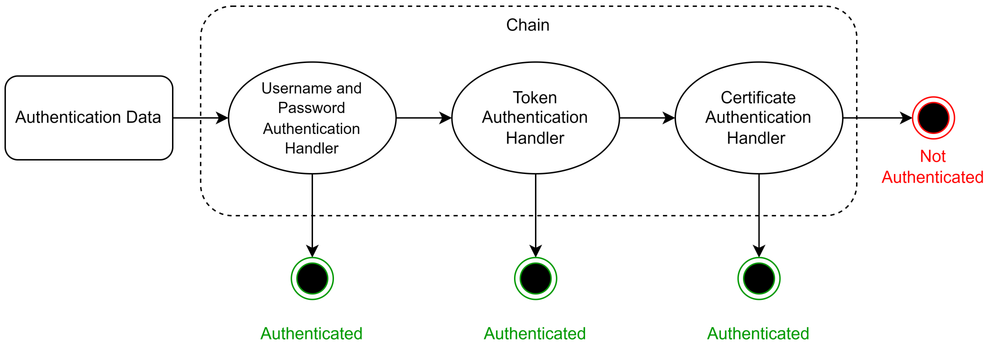
The pattern is useful when a problem being decomposed can be effectively represented as a series of chained objects, in which a request is passed through those objects until one of them is capable of handling it.



For example, let's say that we want to create an authentication system. It should be possible to authenticate using:

1. Username and password
2. Token
3. Certificate

The system should authenticate using the first available method. Such a problem can be presented using the Chain of Responsibility Design Pattern.


Chain
Authentication Data
Username and Password Authentication Handler
Token Authentication Handler
Certificate Authentication Handler
Not Authenticated
Authenticated
Authenticated
Authenticated

2.3.2. Use Cases

Here are some use cases for the Chain of Responsibility Design Pattern:

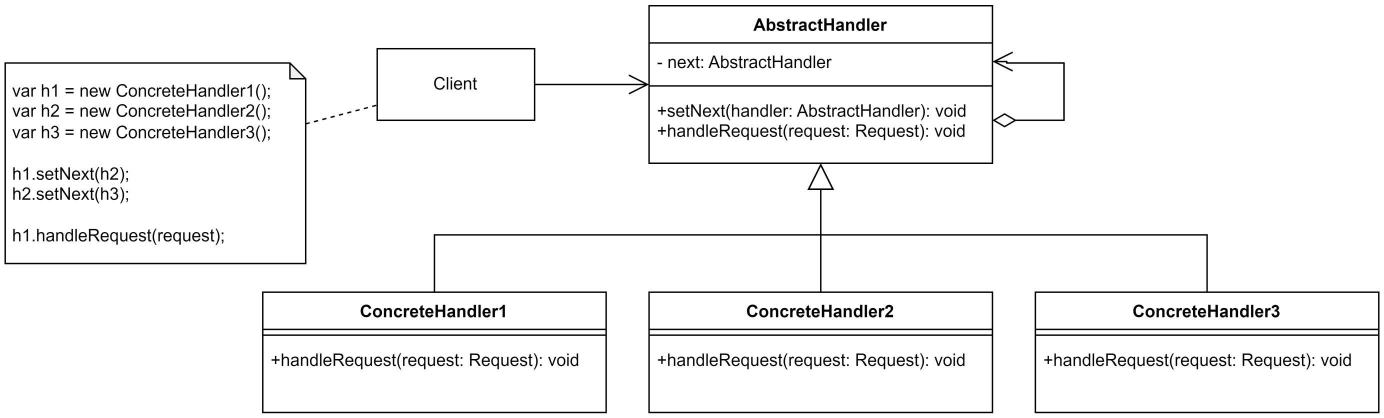
1. Authentication: You can use the Chain of Responsibility Design Pattern to create an authentication system that supports different authentication methods, like username and password-based authentication, token-based authentication, certificate-based authentication, and biometric data based authentication. Each handler in the chain checks if the user can be authenticated with a specific method and passes request to the next handler if authentication method is not supported.
2. Authorization: You can use the Chain of Responsibility Design Pattern to create a chain of authorization handlers that check if a user has the necessary permissions to access a resource. Each handler in the chain can check a specific permission and pass the request to the next handler if the permission is not granted.
3. Exception handling: You can use the Chain of Responsibility Design Pattern to create a chain of exception handlers that handle exceptions based on their type. Each handler in the chain can decide whether to handle the exception or pass it to the next handler in the chain.
4. Validation: You can use the Chain of Responsibility Design Pattern to create a chain of validators that validate a user input based on different criteria. Each validator in the chain can check a specific criterion and pass the input to the next validator if it doesn't meet the criterion.

2.3.3. Structure

The Chain of Responsibility Design Pattern consists of three main components:

1. Handler: An abstract class or an interface that defines the interface for handling requests and has a reference to the next handler in the chain.
2. ConcreteHandler: A concrete implementation of the Handler that will handle the request if it is capable of doing so or will pass the request to the next handler in the chain.
3. Client: The object that creates the chain of handlers and sends requests to the first handler in the chain.

Here is a UML diagram that illustrates the structure of the Chain of Responsibility Design Pattern:


```
var h1 = new ConcreteHandler1();
var h2 = new ConcreteHandler2();
var h3 = new ConcreteHandler3();

h1.setNext(h2);
h2.setNext(h3);

h1.handleRequest(request);
```
Client
AbstractHandler
- next: AbstractHandler
+setNext(handler: AbstractHandler): void
+handleRequest(request: Request): void
ConcreteHandler1
+handleRequest(request: Request): void
ConcreteHandler2
+handleRequest(request: Request): void
ConcreteHandler3
+handleRequest(request: Request): void

2.3.4. Example Code

In the following example, we will create an authentication system using the Chain of Responsibility Design Pattern. We will have three handlers:

1. Username and Password based Authentication Handler
2. Token-based Authentication Handler
3. Certificate-based AuthenticationHandler

The authentication system should authenticate a user based on the first available option. If none of the authentication options successfully authenticate the user, authentication should be unsuccessful.

Let’s start by creating the BaseAuthenticationHandler:

```
public abstract class BaseAuthenticationHandler {
    private BaseAuthenticationHandler nextHandler;

    public abstract boolean handleAuthentication(AuthenticationData authenticationData);

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

    protected boolean handleNext(AuthenticationData authenticationData) {
        if (nextHandler != null) {
            return nextHandler.handleAuthentication(authenticationData);
        }
        return false;
    }
}
```

Then we will implement authentication handlers:

```
public class UsernamePasswordAuthenticationHandler extends BaseAuthenticationHandler {
    @Override
    public boolean handleAuthentication(AuthenticationData authenticationData) {
        System.out.println("Executing username and password based authentication....");

        if (authenticationData.doesSupport(UsernamePasswordAuthenticationData.class)) {
            UsernamePasswordAuthenticationData usernamePasswordAuthenticationData =
authenticationData.toType();

            return usernamePasswordAuthenticationData.getUsername().equals("admin")
                    && usernamePasswordAuthenticationData.getPassword().equals("123");
        }

        return handleNext(authenticationData);
    }
}
```

```
public class TokenAuthenticationHandler extends BaseAuthenticationHandler {
    @Override
    public boolean handleAuthentication(AuthenticationData authenticationData) {
        System.out.println("Executing Token based authentication....");

        if (authenticationData.doesSupport(TokenAuthenticationData.class)) {
            TokenAuthenticationData tokenAuthenticationData = authenticationData.
toType();

            return tokenAuthenticationData.getToken().equals("token200");
        }

        return handleNext(authenticationData);
    }
}

public class CertificateAuthenticationHandler extends BaseAuthenticationHandler {
    @Override
    public boolean handleAuthentication(AuthenticationData authenticationData) {
        System.out.println("Executing Certificate based authentication....");

        if (authenticationData.doesSupport(CertificateAuthenticationData.class)) {
            CertificateAuthenticationData certificateAuthenticationData =
authenticationData.toType();

            return certificateAuthenticationData.getCertificate().equals(
"certificate150");
        }

        return handleNext(authenticationData);
    }
}
```

We will also create a class that will prepare chained authentication handlers:

```
public class AuthenticationHandler {

    private final BaseAuthenticationHandler rootAuthenticationHandler;

    public AuthenticationHandler() {
        var usernamePasswordAuthenticationHandler = new
UsernamePasswordAuthenticationHandler();
        var tokenAuthenticationHandler = new TokenAuthenticationHandler();
        var certificateAuthenticationHandler = new CertificateAuthenticationHandler();

        usernamePasswordAuthenticationHandler.setNextHandler(tokenAuthenticationHandler);
        tokenAuthenticationHandler.setNextHandler(certificateAuthenticationHandler);

        rootAuthenticationHandler = usernamePasswordAuthenticationHandler;
    }

    public boolean handleAuthentication(AuthenticationData authenticationData) {
        System.out.println(
            "Trying to authenticate with " + authenticationData.getClass().
getSimpleName() + "..."
        );

        boolean authenticationResult = rootAuthenticationHandler.handleAuthentication
(authenticationData);

        System.out.println(
            "Authentication was " + (authenticationResult ? "successful" :
"unsuccessful")
        );
        System.out.println();

        return authenticationResult;
    }
}
```

We will then try to authenticate using various methods:

```
var authenticationHandler = new AuthenticationHandler();

authenticationHandler.handleAuthentication(
    new UsernamePasswordAuthenticationData("admin", "123")
);
authenticationHandler.handleAuthentication(
    new TokenAuthenticationData("token200")
);
authenticationHandler.handleAuthentication(
    new CertificateAuthenticationData("certificate150")
);
authenticationHandler.handleAuthentication(
    new BiometricAuthenticationData()
);
```

Which will produce:

```
Trying to authenticate with UsernamePasswordAuthenticationData...
Executing username and password based authentication....
```

```
Authentication was successful

Trying to authenticate with TokenAuthenticationData...
Executing username and password based authentication....
Executing Token based authentication....
Authentication was successful

Trying to authenticate with CertificateAuthenticationData...
Executing username and password based authentication....
Executing Token based authentication....
Executing Certificate based authentication....
Authentication was successful

Trying to authenticate with BiometricAuthenticationData...
Executing username and password based authentication....
Executing Token based authentication....
Executing Certificate based authentication....
Authentication was unsuccessful
```

You can find the source code for this example on https://github.com/dominikcebula/gof-design-patterns/tree/main/src/main/java/com/dominikcebula/edu/design/patterns/behavioral/chain/of/responsibility.

2.3.5. Summary

The Chain of Responsibility Design Pattern is a behavioral pattern that allows you to chain together a series of objects to handle a request in a flexible and extensible way. This pattern is useful when you have a series of objects that can handle a request but don't know which one is capable of handling it. By using the Chain of Responsibility Design Pattern, you can pass the request through the chain of objects until one of them is able to handle it. The structure of the pattern consists of three main components: Handler, ConcreteHandler, and Client. The pattern can be used in various use cases, such as authorization, exception handling and validation.

2.3.6. References

- Erich G, Richard H, Ralph J, John V. Chain of Responsibility. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. October 31, 1994.
- Chain of Responsibility Design Pattern. Refactoring Guru. Accessed May 2, 2023. https://refactoring.guru/design-patterns/chain-of-responsibility
- Chain of Responsibility Design Pattern. Sourcemaking. Accessed May 2, 2023. https://sourcemaking.com/design_patterns/chain_of_responsibility

2.4. Mediator

Not available in the preview.

2.5. Command

Not available in the preview.

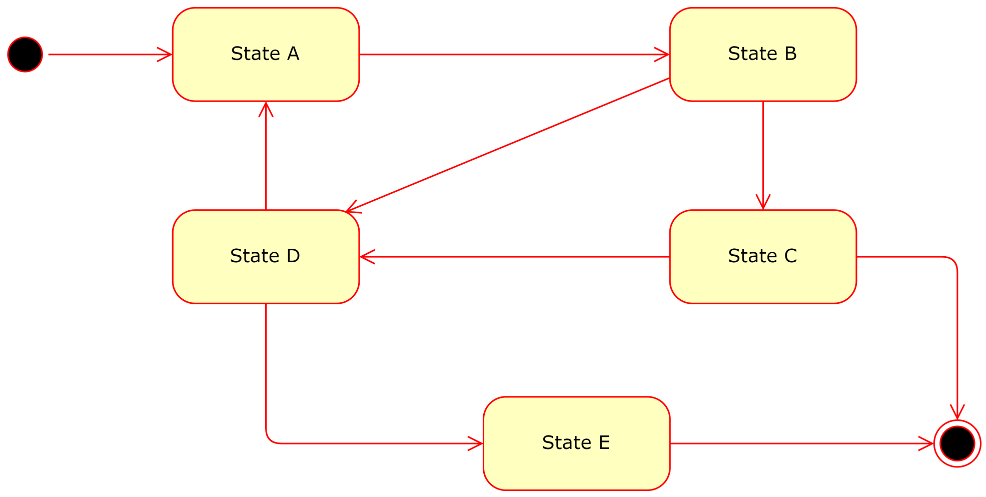
2.6. Memento

Not available in the preview.

2.7. State

2.7.1. Overview

The State design pattern is a behavioral pattern that allows you to implement Finite-State Machine using the Object Oriented Approach.



When a behavior or a business process is described using Finite-State Machine, usually expected behavior and available actions, transitions depend on the current state. The State design pattern offers a structure that allows you to capture state-dependent behaviors in an Object Oriented way. It allows an object to change its behavior when its internal state changes. It involves encapsulating state-specific behaviors into a set of interchangeable state classes. The object delegates state-specific behaviors to a state object. All state objects are represented by a class that implements a common interface. The common interface for state classes allows the main object to switch and delegate state-specific behaviors to the state object. At each time, the object can exist in one of the finite states and can transition between states. Behavior is dependent on the current state. The object usually starts with some initial state, which can be changed during the object lifecycle, which changes its behavior. States can be aware of each other. One state may switch the object state to the next one. Switching the object state is equivalent to transitions in a Finite-State Machine.

2.7.2. Use Cases

The State design pattern is useful when an object has multiple behaviors that need to be switched dynamically depending on the object's state. In addition, the pattern is handy when an object's behavior is complex and difficult to implement using conditional statements to represent state-specific behaviors under each condition.

Potential use cases for the State design pattern involve:

1. Document Editor: A document editor could use the State pattern to manage the different states of the document, such as editing mode, read-only mode, and print preview mode. Each state would have its rules and behaviors that define how the document behaves, such as which buttons are available and what happens when the user tries to edit the document in read-only mode.

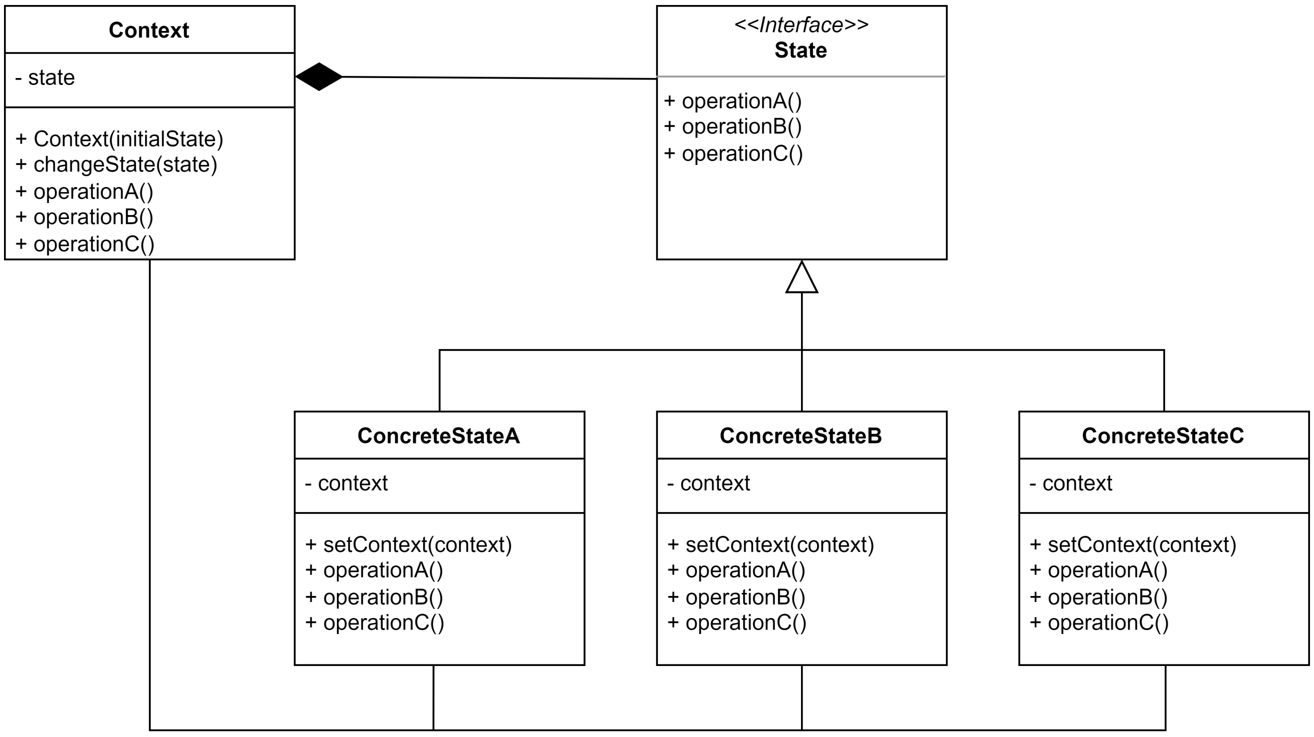
2. Online Shopping Cart: An online shopping cart could use the State pattern to manage the different states of the cart, such as empty and items added. Each state would have its rules and behaviors that define how the cart behaves, such as whether certain buttons are disabled or enabled depending on the current state.
3. Payment System: Payment can exist in multiple states, like initiated, in progress, waiting for confirmation, completed, or failed. Each state defines rules on data, actions, and the next possible state.
4. Music Player: A music player could use the State pattern to manage the different states of the player, such as playing, paused, and stopped. Each state would have its rules and behaviors that define how the player behaves, such as what happens when the user tries to pause the song while the player is stopped.
5. Game Development: A game engine could use the State pattern to manage the different states of the character in the game. The character could exist in multiple states, like standing, walking, or running. Each state is associated with different behavior.

2.7.3. Structure

The State design pattern consists of the following components:

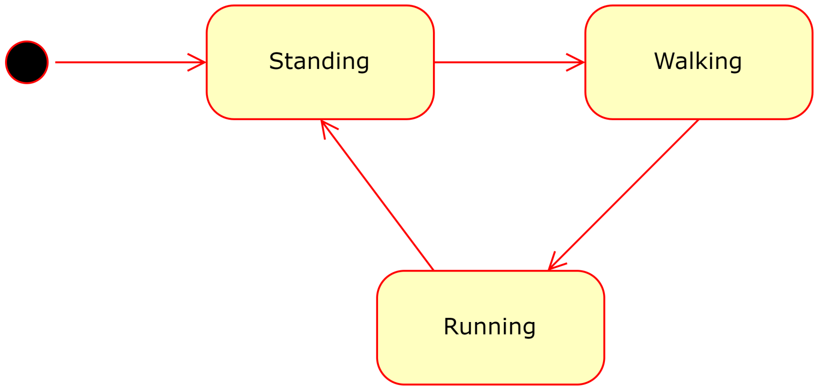
1. Context: The object whose behavior changes based on its internal state.
2. State: The interface that defines the object's behavior based on its state.
3. ConcreteState: The implementation of the State interface that defines the object's behavior for a specific state. States can be aware of each other. One state may switch the context object state to the next one. The state is aware of the context object to initiate state transition.

Here's the UML diagram of the State pattern:



2.7.4. Example Code

In this example code, we will create Object Oriented code that will implement the below State Machine:



The below example will use the following components:

1. Context: will be implemented by GameCharacter class
2. State: will be implemented by CharacterState interface
3. ConcreteStates: will be implemented by following classes:
   a. StandingCharacterState
   b. WalkingCharacterState
   c. RunningCharacterState

First, let’s create an interface that will represent all possible states:

```
public interface CharacterState {
    String getName();

    void move();

    void speak();

    void shoot();

    void toNextState();
}
```

Now, let’s create a class that will encapsulate Game Character behavior for each state.

Standing state:

```
public class StandingCharacterState implements CharacterState {
    private final GameCharacter gameCharacter;

    public StandingCharacterState(GameCharacter gameCharacter) {
        this.gameCharacter = gameCharacter;
    }

    @Override
    public String getName() {
        return "Standing";
    }

    @Override
    public void move() {
        System.out.println("Standing...");
    }

    @Override
    public void speak() {
        System.out.println("Speaking while standing...");
    }

    @Override
    public void shoot() {
        System.out.println("Shooting while standing...");
    }

    @Override
    public void toNextState() {
        gameCharacter.setCharacterState(new WalkingCharacterState(gameCharacter));
    }
}
```

Walking state:

```
public class WalkingCharacterState implements CharacterState {
    private final GameCharacter gameCharacter;

    public WalkingCharacterState(GameCharacter gameCharacter) {
        this.gameCharacter = gameCharacter;
    }

    @Override
    public String getName() {
        return "Walking";
    }

    @Override
    public void move() {
        System.out.println("Walking...");
    }

    @Override
    public void speak() {
        System.out.println("Speaking while walking...");
    }

    @Override
    public void shoot() {
        System.out.println("Shooting while walking...");
    }

    @Override
    public void toNextState() {
        gameCharacter.setCharacterState(new RunningCharacterState(gameCharacter));
    }
}
```

Running state:

```
public class RunningCharacterState implements CharacterState {
    private final GameCharacter gameCharacter;

    public RunningCharacterState(GameCharacter gameCharacter) {
        this.gameCharacter = gameCharacter;
    }

    @Override
    public String getName() {
        return "Running";
    }

    @Override
    public void move() {
        System.out.println("Running...");
    }

    @Override
    public void speak() {
        System.out.println("Speaking while running...");
    }

    @Override
    public void shoot() {
        System.out.println("Shooting while running...");
    }

    @Override
    public void toNextState() {
        gameCharacter.setCharacterState(new StandingCharacterState(gameCharacter));
    }
}
```

Now, we will create a GameCharacter that will act as a Context:

```
public class GameCharacter {

    private CharacterState characterState = new StandingCharacterState(this);

    public void setCharacterState(CharacterState characterState) {
        this.characterState = characterState;
    }

    public CharacterState getCharacterState() {
        return characterState;
    }

    public void move() {
        characterState.move();
    }
    public void speak() {
        characterState.speak();
    }

    public void shoot() {
        characterState.shoot();
    }

    public void toNextState() {
        characterState.toNextState();
    }
}
```

We can use the above code in the following way:

```
GameCharacter gameCharacter = new GameCharacter();

System.out.println(
    "Game character state = " + gameCharacter.getCharacterState().getName()
);
gameCharacter.move();
gameCharacter.speak();
gameCharacter.shoot();

gameCharacter.toNextState();
System.out.println();

System.out.println(
    "Game character state = " + gameCharacter.getCharacterState().getName()
);
gameCharacter.move();
gameCharacter.speak();
gameCharacter.shoot();

gameCharacter.toNextState();
System.out.println();

System.out.println(
    "Game character state = " + gameCharacter.getCharacterState().getName()
);
gameCharacter.move();
gameCharacter.speak();
gameCharacter.shoot();

gameCharacter.toNextState();
System.out.println();
System.out.println(
    "Game character state = " + gameCharacter.getCharacterState().getName()
);
gameCharacter.move();
gameCharacter.speak();
gameCharacter.shoot();
```

The above code will create the following output:

```
Game character state = Standing
Standing...
Speaking while standing...
Shooting while standing...

Game character state = Walking
Walking...
Speaking while walking...
Shooting while walking...

Game character state = Running
Running...
Speaking while running...
Shooting while running...

Game character state = Standing
Standing...
Speaking while standing...
Shooting while standing...
```

Notice how GameCharacter changes its behavior when transiting between states. Each state involves different behavior without increasing the context object's complexity.

You can find the source code for this example on https://github.com/dominikcebula/gof-design-patterns/tree/main/src/main/java/com/dominikcebula/edu/design/patterns/behavioral/state.

2.7.5. Limitations

Since the State design pattern requires each state to be encapsulated in a class that implements a common interface, it can easily lead to violating Interface Segregation Principle (ISP) and Liskov Substitution Principle (LSP). If you would observe in your code something like this:

```
class ConcreteState implements State {
    @Override
    public void operationA() {
        throw new IllegalStateException("this state does not support operationA");
    }

    @Override
    public void operationB() {
        // empty method, this state will not support this operation
    }

    // implementation details
}
```

Then it might mean that structure offered by this pattern does not match your needs.

2.7.6. Summary

The State design pattern is a behavioral pattern that allows you to implement Finite-State Machine using the Object Oriented Approach. The pattern delegates state-specific behaviors to state objects. It involves three components: Context, State, ConcreteState. The State design pattern is useful when an object has multiple behaviors that need to be switched dynamically based on the object's state.

2.7.7. References

1. Erich G, Richard H, Ralph J, John V. State. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. October 31, 1994.
2. State Design Pattern. Refactoring Guru. Accessed April 10, 2023. https://refactoring.guru/design-patterns/state
3. State Design Pattern. Sourcemaking. Accessed April 10, 2023. https://sourcemaking.com/design_patterns/state

2.8. Interpreter

Not available in the preview.

2.9. Observer

Not available in the preview.

2.10. Visitor

Not available in the preview.

2.11. Iterator

Not available in the preview.

Chapter 3. Structural Patterns

3.1. Facade

Not available in the preview.

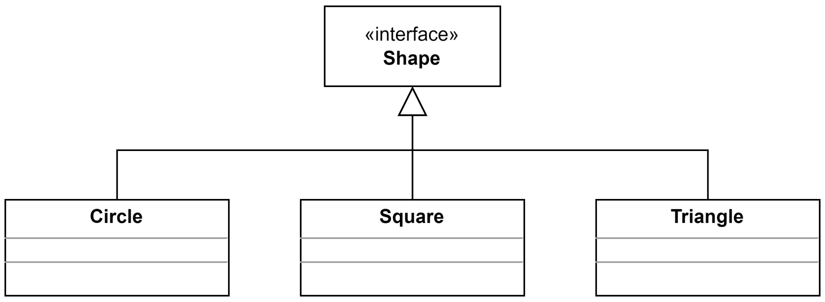
3.2. Bridge

3.2.1. Overview

The Bridge Design Pattern is a structural design pattern that solves the issue of exponential growth of classes that may occur when decomposing the problem being solved into objects.

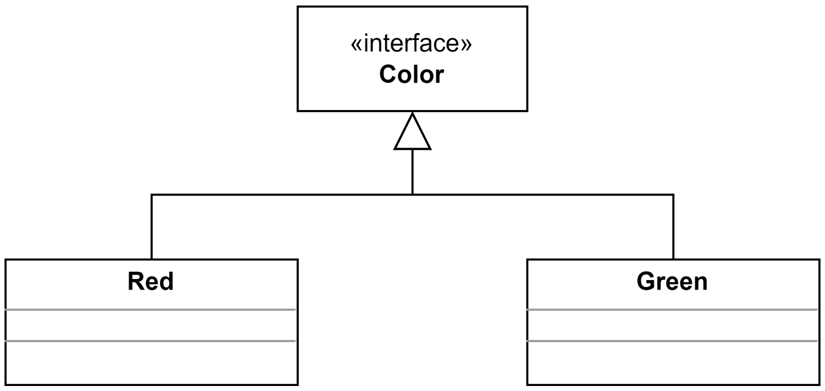
Imagine that you need to model the following shapes:

1. Circle
2. Square
3. Triangle



And each shape has the following colors available:

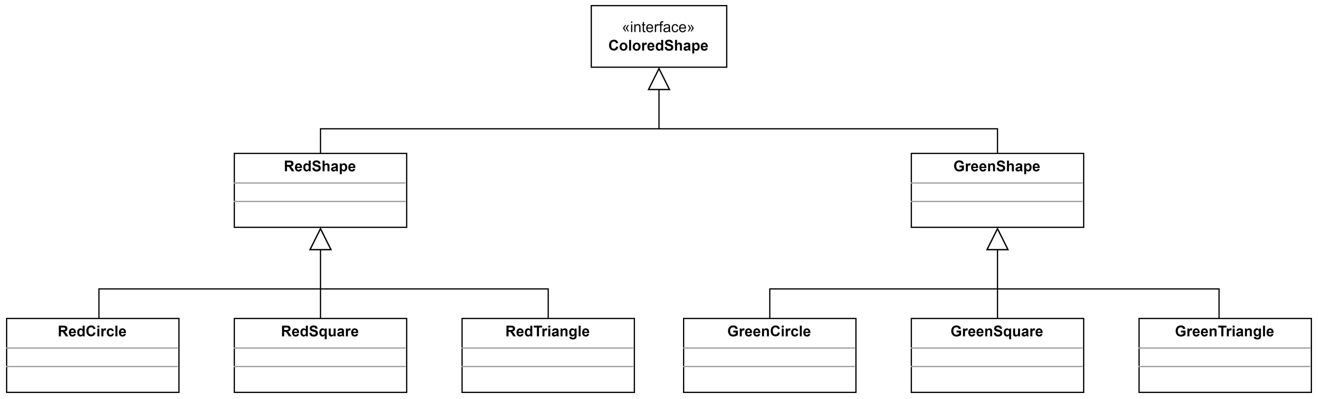
1. Red
2. Green



If we would try to create a class for each shape, we would end up with 3 x 2 = 6 classes:

1. RedCircle
2. RedSquare
3. RedTriangle
4. GreenCircle
5. GreenSquare

6. GreenTriangle

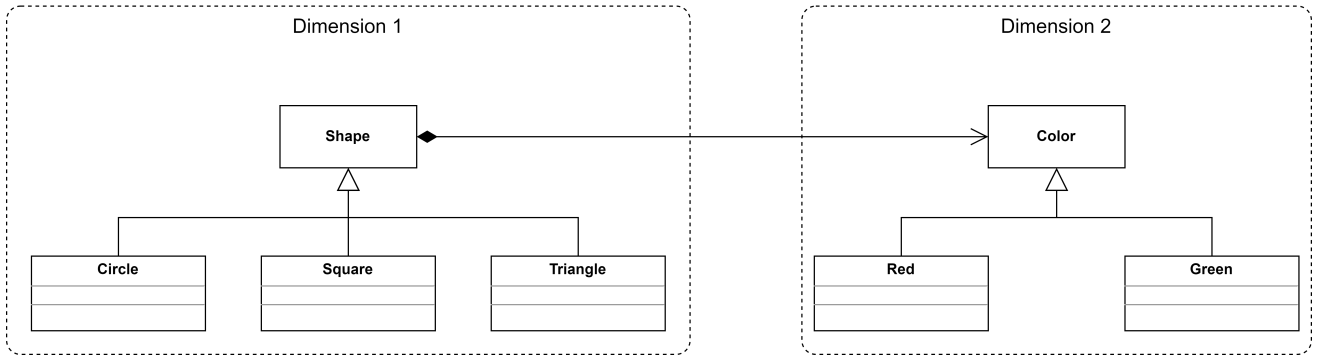


It is easy to observe that such an approach will not scale and will result in many code duplicates.

We can also observe that this problem has two separate dimensions:

1. Shape
2. Color

The Bridge design pattern solves the exponential growth of classes by modeling each dimension separately, allowing for those two dimensions to cooperate using composition.



Two dimensions introduced by the Bridge design pattern are named:

1. Control layer – also called Abstraction or Interface
2. Platform Layer – also called Implementation

Note that over here, we do not use Abstraction, Interface, or Implementation in an OOP sense.

The other example of a multi-dimension problem would be the implementation of UI for multiple platforms.

The first dimension would be UI elements:

1. Button.
2. Checkbox.

The second dimension would be a platform on which each can run:

1. Linux.
2. Windows.
3. Mac.

If we would try to decompose this problem in a classic way, we would get:

1. LinuxButton.
2. LinuxCheckbox.
3. WindowsButton.
4. WindowsCheckbox.
5. MacButton.
6. MacCheckbox.

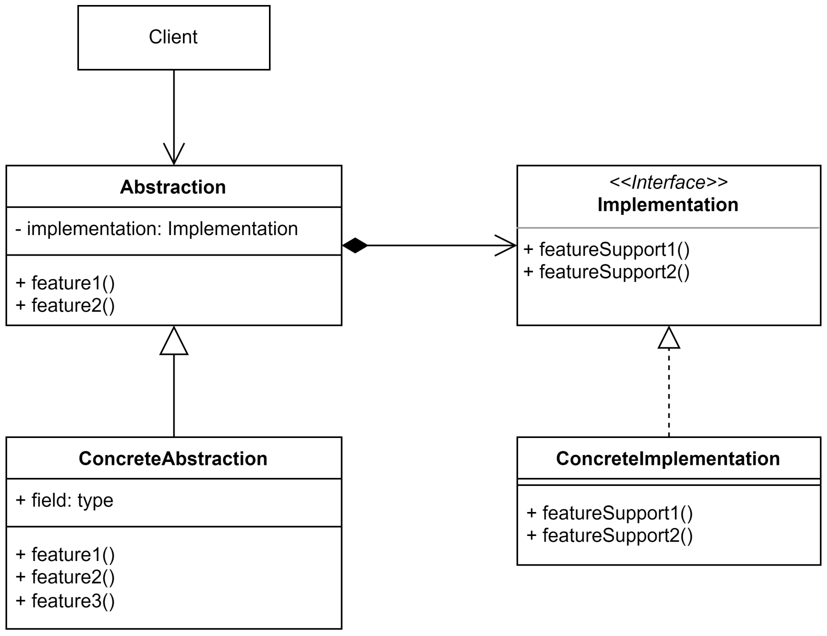
We can decompose this problem using separated dimensions using the Bridge design pattern.

3.2.2. Use Cases

1. Multi-Platform UI Components Framework: In a GUI application, you may have different types of controls (such as buttons, checkboxes, and text boxes) that need to work with different operating systems (such as Linux, Windows, Mac). The Bridge pattern can be used to separate the control's functionality from its implementation, allowing you to easily add support for new operating systems.
2. Universal OS Components: You may have different OS components, like Process, Thread, Socket, that need to work on different platforms, like Linux, Windows, Mac. The Bridge pattern can be used to separate the logical functionality from its implementation for a specific platform.
3. Logging: You may want to log messages to different destinations (such as a file, a database, or a remote server) using different formats (such as plain text, JSON, or XML). The Bridge pattern can be used to separate the logging logic from its implementation, allowing you to easily add support for new destinations and formats.
4. Platform Independent Mobile Application: mobile application that has a consistent user interface and functionality across both Android and iOS platforms. The Bridge design pattern is used to separate the abstraction (the user interface) from the implementation (the platform-specific code).

3.2.3. Structure

The Bridge pattern has two main components: the Abstraction and the Implementation. The Abstraction (control layer) defines the high-level functionality and maintains a reference to an object of type Implementation (platform layer). The Abstraction provides a control logic, we may say that is orchestrates the Implementation. The Implementation is an abstract class that defines the low-level functionality, and its concrete subclasses provide the actual implementation of the abstraction's methods. The Abstraction uses the Implementation through a reference to the Implementation.



3.2.4. Example Code

In the below example, we will use the bridge design pattern to create the class structure for UI Framework that will provide two components:

1. Button
2. Checkbox

that can work on three platforms:

1. Linux
2. Windows
3. Mac

We will use the following parts of the bridge design pattern:

1. The Abstraction (control layer): will be implemented by UI Components (Button and Checkbox)
2. Implementation (platform layer): will be implemented by Platform specific code (LinuxPlatform, WindowsPlatform, MacPlatform)

First, we will create the base Component class, that will hold a reference to the specific Platform:

```
abstract class Component {
    final Platform platform;

    Component(Platform platform) {
        this.platform = platform;
    }
}
```

Then, we will create two components, Button:

```
public class Button extends Component {
    public Button(Platform platform) {
        super(platform);
    }

    public void click() {
        System.out.println("Button was clicked.");
        platform.updateUI();
    }
}
```

and Checkbox:

```
public class Checkbox extends Component {
    public Checkbox(Platform platform) {
        super(platform);
    }

    public void check() {
        System.out.println("Checkbox was checked.");
        platform.updateUI();
    }

    public void uncheck() {
        System.out.println("Checkbox was unchecked.");
        platform.updateUI();
    }
}
```

Then, we will create each Platform on which the component can render:

```
public class LinuxPlatform implements Platform {
    @Override
    public void updateUI() {
        System.out.println("Updating UI on Linux Platform.");
    }
}

public class WindowsPlatform implements Platform {
    @Override
    public void updateUI() {
        System.out.println("Updating UI on Windows Platform.");
    }
}

public class MacPlatform implements Platform {
    @Override
    public void updateUI() {
        System.out.println("Updating UI on Mac Platform.");
    }
}
```

Now, we can use the above code in the following way:

```
Platform linuxPlatform = new LinuxPlatform();
Platform windowsPlatform = new WindowsPlatform();
Platform macPlatform = new MacPlatform();

Button buttonOnLinux = new Button(linuxPlatform);
Button buttonOnWindows = new Button(windowsPlatform);

buttonOnLinux.click();
buttonOnWindows.click();

Checkbox checkboxOnLinux = new Checkbox(linuxPlatform);
Checkbox checkboxOnMac = new Checkbox(macPlatform);

checkboxOnLinux.check();
checkboxOnLinux.uncheck();
checkboxOnMac.check();
```

Which will produce the following results:

```
Button was clicked.
Updating UI on Linux Platform.
Button was clicked.
Updating UI on Windows Platform.
Checkbox was checked.
Updating UI on Linux Platform.
Checkbox was unchecked.
Updating UI on Linux Platform.
Checkbox was checked.
Updating UI on Mac Platform.
```

Notice how in the above code, the bridge design pattern is used to decompose the Multi-Platform UI Components Framework problem in a way that allows us to avoid exponential growth of classes issue.

You can find the source code for this example on https://github.com/dominikcebula/gof-design-patterns/tree/main/src/main/java/com/dominikcebula/edu/design/patterns/structural/bridge.

3.2.5. Summary

The Bridge Design Pattern solves the exponential growth of classes problems. It is achieved by modeling dimensions as the Abstraction (control layer) and the Implementation (platform layer). Abstraction (control layer) holds a reference to an object from the Implementation (platform layer) and uses it through composition. The pattern is a good fit for problems in which you can observe distinct dimensions, which otherwise will result in N x M amount of classes.

3.2.6. References

1. Erich G, Richard H, Ralph J, John V. Bridge. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. October 31, 1994.
2. Bridge Design Pattern. Refactoring Guru. Accessed April 29, 2023. https://refactoring.guru/design-patterns/bridge
3. Bridge Design Pattern. Sourcemaking. Accessed April 29, 2023. https://sourcemaking.com/design_patterns/bridge

3.3. Decorator

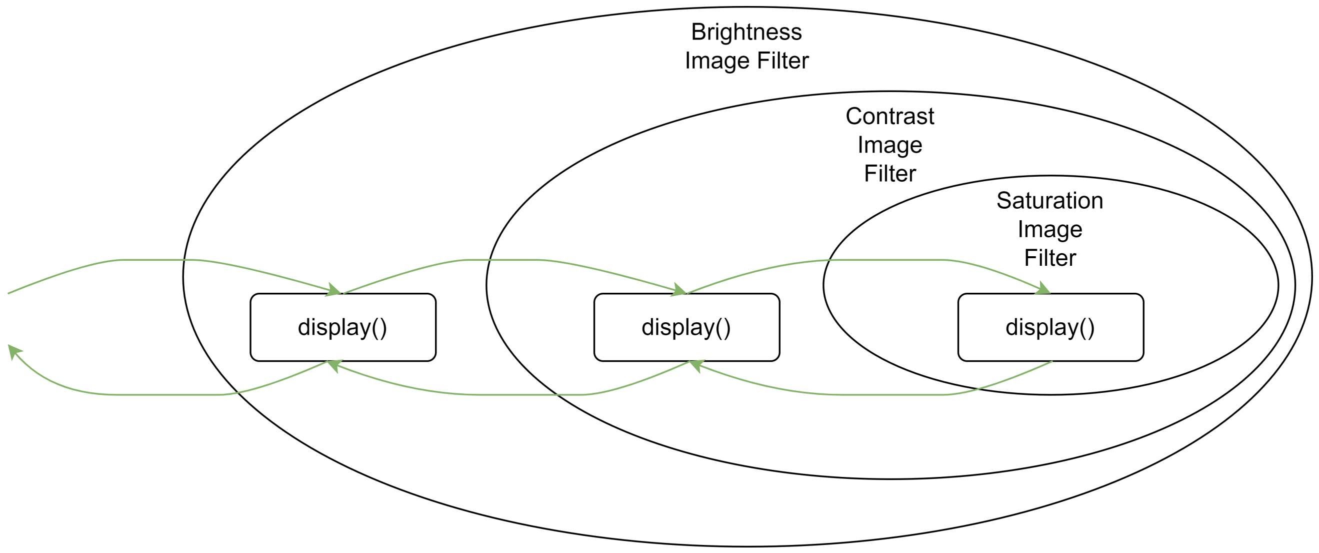
3.3.1. Overview

The Decorator design pattern is a structural pattern that allows you to add functionality to an object dynamically without changing its original structure. This is achieved by wrapping objects around other objects, often multiple times. Usage of this pattern allows you to write the code like this:

```
var filteredImage = new SaturationFilter(
        new ContrastFilter(
                new BrightnessFilter(
                      image
                )
        )
);
```

Using the above structure, you gain the flexibility of being able to easily change which objects are wrapped around each other and in what configuration.

We can visualize this pattern in the following way:



This pattern involves creating a decorator class that wraps around the original object and adds new behaviors or functionalities. This pattern offers better flexibility and extendibility compared to implementing the same functionality using inheritance. The main idea behind this pattern is to expose expected behaviors using a same interface that is implemented by the concrete component and the decorating component. The concrete component delivers some specific functionality, while the decorating components allow the wrapping of the original object, often multiple times. Each additional wrapper adds additional behavior. The client depends on the uniform interface and does not know the implementation details of how objects are chained together. When functionality needs to be extended, objects are chained differently. However, the objects structure or the way that the client interacts with objects does not need to change. That way, the Decorator design pattern offers flexibility and extendibility.

3.3.2. Use Cases

The decorator design pattern is a good fit whenever you need a flexible structure of chaining together related actions in different configurations. Example use cases:

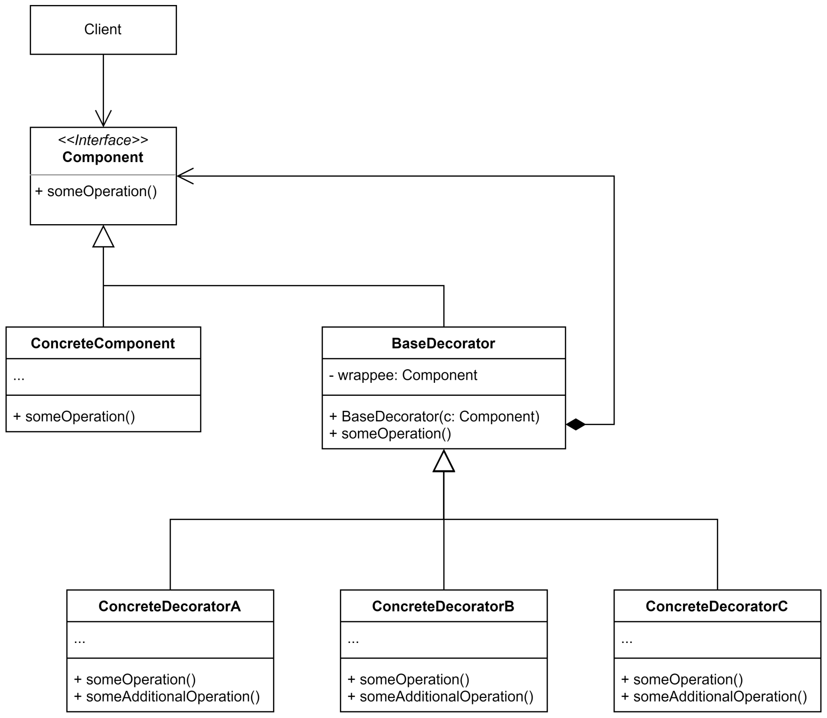
1. Image processing: Ability to reach flexibility on adding new filters, such as brightness, contrast, or saturation, to images, without modifying object structure. Each additional filter is a decorator which adds image processing functionality. The client interacts with the uniform interface. Details on the exact filters that are used are hidden. When filters change, the client does not need to change.
2. File services with added functionality: Ability to implement flexible objects structure that can save, compress and encrypt data. In some cases, we want to save the plain text file, in some cases only compressed files, and in some cases, compressed and encrypted files. The decorator design pattern is used to chain file content modifiers which can be chained differently without changing the client.
3. Email service: Adding encryption, digital signatures, or compression to emails sent by the system without modifying the original object structure.
4. Document processing: Adding headers, footers, page numbers, or watermarks to a document without changing the original object structure.
5. Reporting system: Adding new formatting options, such as font size, color, or style, to reports generated by the system, without altering the original object structure.

3.3.3. Structure

The Decorator pattern consists of four main components:

1. Component: An interface defining the common operations for the original object and its decorators. The client will depend on this interface and will use exposed operations through it.
2. Concrete Component: A class that represents the original object that we want to decorate.
3. Base Decorator: An abstract class that wraps around the original object. It delegates operations to the wrapped object. It looks at the wrapped object through a uniform interface. That way, it can delegate to the final Component or next Decorator.
4. Concrete Decorator: A class that extends the Decorator class and adds specific functionalities to the original object.

Here is a UML diagram that shows the structure of the Decorator pattern:



3.3.4. Example Code

In the below example, we will implement the Image Processing use case using the Decorator Design Pattern.

In this example, we will have the following components:

1. Component: the uniform interface will be implemented by Image interface.
2. Concrete Component: will be implemented by FileImage.
3. Base Decorator: will be implemented by ImageFilter.
4. Concrete Decorator: will be implemented by BrightnessFilter, ContrastFilter, SaturationFilter.

First, let's implement the Component interface represented by Image interface:

```java
public interface Image {
    void display();
}
```

Then, let's implement the Concrete Component represented by FileImage:

```java
public class FileImage implements Image {
    private final String fileName;

    public FileImage(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void display() {
        System.out.println("Displaying " + fileName);
    }
}
```

Now, let’s implement the Base Decorator represented by ImageFilter:

```
abstract class ImageFilter implements Image {
    protected Image image;

    public ImageFilter(Image image) {
        this.image = image;
    }

    @Override
    public void display() {
        image.display();
    }
}
```

Finally, we will implement Concrete Decorators:

```
public class BrightnessFilter extends ImageFilter {
    public BrightnessFilter(Image image) {
        super(image);
    }

    @Override
    public void display() {
        System.out.println("Applying brightness filter");
        super.display();
    }
}

public class ContrastFilter extends ImageFilter {
    public ContrastFilter(Image image) {
        super(image);
    }

    @Override
    public void display() {
        System.out.println("Applying contrast filter");
        super.display();
    }
}

public class SaturationFilter extends ImageFilter {
    public SaturationFilter(Image image) {
        super(image);
    }

    @Override
    public void display() {
        System.out.println("Applying saturation filter");
        super.display();
    }
}
```

We can use the above code in the following way:

```
Image image = new FileImage("image.jpg");

Image filteredImage1 = new SaturationFilter(
        new ContrastFilter(
                new BrightnessFilter(image)
        )
);

Image filteredImage2 = new ContrastFilter(
        new BrightnessFilter(image)
);

filteredImage1.display();
filteredImage2.display();
```

Which will produce:

```
Applying saturation filter
Applying contrast filter
Applying brightness filter
Displaying image.jpg

Applying contrast filter
Applying brightness filter
Displaying image.jpg
```

Notice how easy it is to change filters applied when the image is displayed. Since the Client depends on the uniform interface, which under the hood can be implemented by the concrete image or filters that delegate to other filters, we can easily implement different use cases like:

1. Displaying image without any filters.
2. Displaying image with brightness filter.
3. Displaying image with contrast filter.
4. Displaying image with brightness and saturation filter.
5. Displaying image with brightness and contrast filter.
6. Displaying image with brightness, contrast, and saturation filter.

For each use case, we only need to change how filters are chained.

You can find the source code for this example on https://github.com/dominikcebula/gof-design-patterns/tree/main/src/main/java/com/dominikcebula/edu/design/patterns/structural/decorator.

3.3.5. Summary

The Decorator design pattern is a way of adding functionality to an object without changing its original structure. It involves creating a decorator class that wraps around the original object and adds new behaviors. The main concept is to have a uniform interface that is implemented by the concrete component and the decorator components. The concrete component delivers specific functionality, while the decorator components wrap the original object, often multiple times, and add additional behavior. The client depends on the uniform interface and does not need to know the implementation details. When functionality needs to be extended, objects are chained differently, but the object structure and the way the client interacts with objects does not need to change. The main benefit of this pattern is flexibility and extendibility.

3.3.6. References

1. Erich G, Richard H, Ralph J, John V. Decorator. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. October 31, 1994.
2. Decorator Design Pattern. Refactoring Guru. Accessed April 14, 2023. https://refactoring.guru/design-patterns/decorator
3. Decorator Design Pattern. Sourcemaking. Accessed April 14, 2023. https://sourcemaking.com/design_patterns/decorator

3.4. Composite

Not available in the preview.

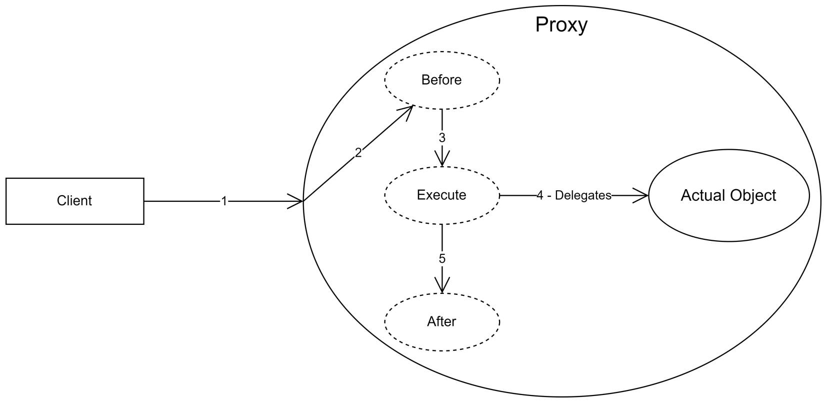
3.5. Proxy

3.5.1. Overview

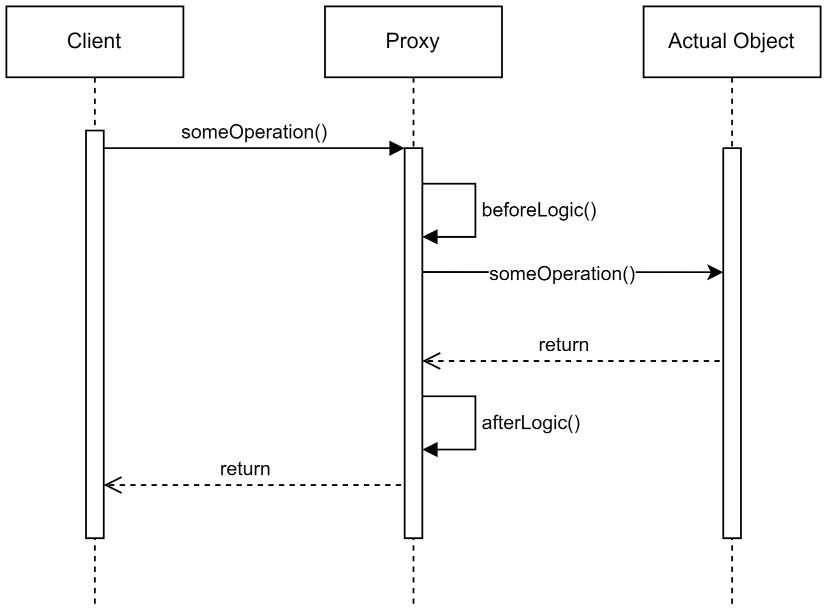
The Proxy design pattern is a structural design pattern that provides a substitute or a placeholder for the original object. The proxy wraps the original object, and each access to the original object always goes through the proxy. Since the original object and the proxy implement the same interface, the client is unable to distinguish between the original object and the proxy. The proxy allows you to implement some additional logic before and after the original method is executed. It is often used to implement cross-cutting concerns like additional logging or security.

Proxy usually adds before and after logic that is executed around the actual method that is invoked. Proxy delegates actual method execution to the object that is being proxied. Implementing both before and after logic is not mandatory. Some proxies might implement only one of them.

A proxy hides the actual object, keeps a reference to it, and the client invokes the actual object through the proxy:



Here is the sequence of operations involved with actual object execution via proxy:



3.5.2. Use Cases

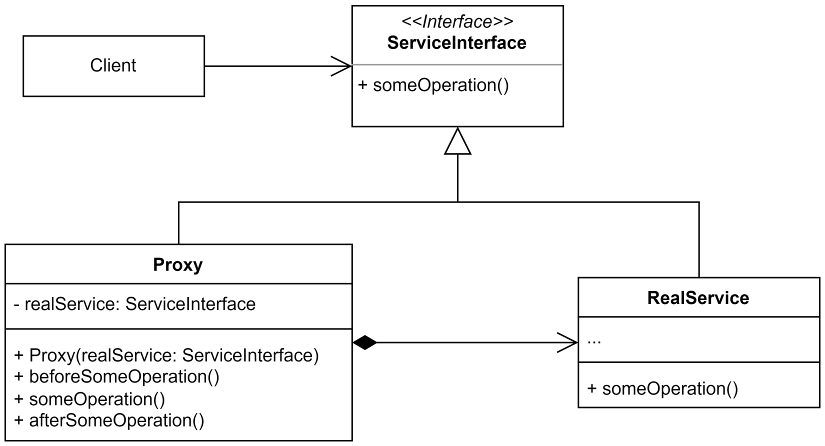
1. Security Access Control proxies: When an object needs to be protected from unauthorized access, a protection proxy can be used to control access to the object. The protection proxy can check the credentials of the client before allowing access to the object.
2. Lazy Initialization Virtual proxies: When an object is expensive to create or initialize, a lazy initializing virtual proxy can be used to create a lightweight representation of the object. The virtual proxy can defer the creation of the real object until it is actually needed, thus improving performance.
3. Logging Proxy: When we need additional logging whenever the method on the original object is executed, Logging Proxy should be considered. In this approach, the client interacts with Logging Proxy, which pretends to be the original object. Logging Proxy logs each method invocation, often on the tracing logging level. This can be useful for production support purposes.
4. Caching Proxy: When method results is expensive to get, and at the same time, method results can be cached, Caching Proxy should be considered. The idea here is to wrap the original object within a proxy object that will save the result of method invocation when the method is invoked for the first time. When the method is invoked for the second time, pre-fetched results are taken from the cache.
   - **Warning** – cache usage always increases the complexity, as sooner or later, the cache needs to be properly invalidated. Failing to properly invalidate the cache is often a source of hard-to-trace bugs.
5. Remote proxies: When an object is located in a remote location and needs to be accessed over the network, a remote proxy can be used to provide a local representation of the object, which is responsible for hiding all details related to network communication with an object located on another machine. Using this approach, an object that is located in a remote location is used as it would be a local object.
   - **Warning** – although this approach does simplify the code by hiding all network communication details, it can lead to performance or service availability issues. Using an object through the network is always different than using a local object within the same machine / jvm / memory. In-memory calls are very different when compared to over-the-network calls. With the network, you need to consider latency and possible communication errors. Using the Remote Proxy approach, it is easy to forget about this and treat this object as a local object. When viewing a remote object as a local object, we can forget that those are actually network calls, and a call to each method might not be cheap, which in the end, will increase the chattiness of the application. The same applies to availability. When communicating with a remote object, we often need to embed communication error handling into the business logic flow. Logic, like start releasing the inventory when the call to the payment system fails, needs to be possible. If the proxy would hide error details and would only implement simple retries, having this logic would not be possible.

3.5.3. Structure

The Proxy design pattern consists of several components:

1. Service Interface: The interface that defines the operations exposed by the service being proxied. The real service and the proxy will share and implement operations from this interface.
2. Real Service: The real object that is being proxied.
3. Proxy: The object that acts as a substitute for the real service. The proxy has the same interface as the real service and delegates requests to the real service. The proxy allows adding before and after additional logic whenever a method on the real service is executed.

Here's a UML diagram that illustrates the structure of the Proxy design pattern:



3.5.4. Example Code

In the below example, we will use the proxy design pattern to implement a security proxy for an object that will model bank account operations.

Additionally, we will have an authentication token that holds granted authorities. The security proxy will verify bank account operation eligibility based on authorities in the authentication token.

In the example, we will use the following components:

1. Service Interface: will be implemented by BankAccount interface. This interface will expose the following operations: deposit, withdraw, getBalance.
2. Real Service: Will be implemented by BankAccountImpl. This object will provide logic for the following operations: deposit, withdraw, getBalance.
3. Proxy: Will be implemented by BankAccountSecurityProxy. This proxy will provide security cross-cutting concerns. It will check the eligibility of the following operations: deposit, withdraw, getBalance.

Let's start by implementing BankAccount interface:

```java
public interface BankAccount {
    void deposit(Money amount);

    void withdraw(Money amount);

    Money getBalance();
}
```

Now, we will implement real bank account operation under BankAccountImpl:

```java
public class BankAccountImpl implements BankAccount {
    private Money balance = Money.of(0, "USD");

    @Override
    public void deposit(Money amount) {
        balance = balance.add(amount);
    }

    @Override
    public void withdraw(Money amount) {
        if (balance.isGreaterThanOrEqualTo(amount))
            balance = balance.subtract(amount);
        else
            throw new IllegalStateException(
                    "Unable to withdraw amount " + amount
                    + " because amount exceeds current balance " + balance
            );
    }

    @Override
    public Money getBalance() {
        return balance;
    }
}
```

Prior implementing the security proxy, we will implement the other required objects:

```java
public record AuthenticationToken(String name, Set<Authority> authorities) {
    public boolean hasAuthority(Authority authority) {
        return authorities.contains(authority);
    }
}

public enum Authority {
    ALLOWED_DEPOSIT,
    ALLOWED_WITHDRAW,
    ALLOWED_GET_BALANCE,
}
```

Now, we can implement BankAccountSecurityProxy that will check eligibility of each bank operation, based on AuthenticationToken and Authority:

```
public class BankAccountSecurityProxy implements BankAccount {
    private final BankAccount bankAccount;
    private final AuthenticationToken authenticationToken;

    public BankAccountSecurityProxy(BankAccount bankAccount, AuthenticationToken
authenticationToken) {
        this.bankAccount = bankAccount;
        this.authenticationToken = authenticationToken;
    }

    @Override
    public void deposit(Money amount) {
        checkIfAllowedToDeposit();
        bankAccount.deposit(amount);
    }

    @Override
    public void withdraw(Money amount) {
        checkIfAllowedToWithdraw();
        bankAccount.withdraw(amount);
    }

    @Override
    public Money getBalance() {
        checkIfAllowedGetBalance();
        return bankAccount.getBalance();
    }

    private void checkIfAllowedToDeposit() {
        if (!authenticationToken.hasAuthority(ALLOWED_DEPOSIT))
            throw new IllegalStateException(
              "User is not allowed to deposit money using token " + authenticationToken
            );
    }

    private void checkIfAllowedToWithdraw() {
        if (!authenticationToken.hasAuthority(ALLOWED_WITHDRAW))
            throw new IllegalStateException(
                "User is not allowed to withdraw money using token " + authenticationToken
            );
    }

    private void checkIfAllowedGetBalance() {
        if (!authenticationToken.hasAuthority(ALLOWED_GET_BALANCE))
            throw new IllegalStateException(
              "User is not allowed to get balance using token " + authenticationToken
            );
    }
}
```

Notice, how the above code implements the same interface as the real bank account object, but prior delegating the operation to the real bank account object, it executes checkIfAllowedToDeposit, checkIfAllowedToWithdraw, checkIfAllowedGetBalance first.

Now, we can use the above code as following:

```
AuthenticationToken authenticationToken = new AuthenticationToken(
    "john", Set.of(ALLOWED_DEPOSIT, ALLOWED_GET_BALANCE)
);

BankAccount bankAccount = new BankAccountSecurityProxy(
        new BankAccountImpl(),
        authenticationToken
);

System.out.println("Depositing money...");
bankAccount.deposit(Money.of(500, "USD"));
bankAccount.deposit(Money.of(300, "USD"));

System.out.println("Getting current balance...");
Money currentBalance = bankAccount.getBalance();
System.out.println("Current balance = " + currentBalance);

System.out.println("Withdrawing money...");
bankAccount.withdraw(Money.of(100, "USD"));
```

The above code will produce the following output:

```
Depositing money...
Getting current balance...
Current balance = USD 800.00
Withdrawing money...
Exception in thread "main" java.lang.IllegalStateException: User is not allowed to
```

```
withdraw money using token AuthenticationToken[name=john,
authorities=[ALLOWED_GET_BALANCE, ALLOWED_DEPOSIT]]
```

Notice how BankAccountSecurityProxy throws an exception on withdraw operation. This is because used AuthenticationToken does not have ALLOWED_WITHDRAW authority assigned.

You can find the source code for this example on https://github.com/dominikcebula/gof-design-patterns/tree/main/src/main/java/com/dominikcebula/edu/design/patterns/structural/proxy.

3.5.5. Alternatives

The Proxy design pattern can be implemented manually, like in the example above, however, you should also consider the following alternatives:

1. JDK Dynamic Proxy - https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/reflect/Proxy.html
2. CGLIB Proxy using CGLIB Enhancer - https://github.com/cglib/cglib
3. Aspect-oriented programming with AspectJ - https://www.eclipse.org/aspectj/
4. Aspect Oriented Programming with Spring - https://docs.spring.io/spring-framework/reference/core/aop.html

3.5.6. Summary

The Proxy design pattern is a structural pattern that provides an object that acts as a substitute for the original object by wrapping it. The proxy and the original object implement the same interface, and each access to the original object goes through the proxy. The proxy enables adding extra logic before and after executing the original method, typically for implementing cross-cutting concerns like logging, caching, or security. The pattern consists of the following key components: Service Interface, Real Service, and the Proxy itself.

3.5.7. References

1. Erich G, Richard H, Ralph J, John V. Proxy. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. October 31, 1994.
2. Proxy Design Pattern. Refactoring Guru. Accessed April 16, 2023. https://refactoring.guru/design-patterns/proxy
3. Proxy Design Pattern. Sourcemaking. Accessed April 16, 2023. https://sourcemaking.com/design_patterns/proxy

3.6. Adapter

Not available in the preview.

3.7. Flyweight

Not available in the preview.

Chapter 4. Creational Patterns

4.1. Abstract Factory

Not available in the preview.

4.2. Factory Method

Not available in the preview.

4.3. Builder

Not available in the preview.

4.4. Singleton

Not available in the preview.

4.5. Prototype

Not available in the preview.

Chapter 5. Summary

Not available in the preview.