CREATIONAL DESIGN PATTERN — BUILDER PATTERN

Builder Design Pattern In Java

Build Your Class Objects in the Easiest Way

Vikram Gupta · Follow

Published in Javarevisited · 5 min read · Feb 8, 2023

In this article, we will see what is the intent of the **Builder Design Pattern**, what problems it solves, and its applicability.

There are different creational design patterns and they have their intent and applicability.

Creational design patterns ***provide various object creation mechanisms, which increase flexibility and reuse of existing code: —***

1. **Factory Method** — A creational design pattern ***that provides an Interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.***
2. **Abstract Factory Method** — A creational design pattern ***that lets you produce families of related objects without specifying their concrete classes.***
3. **Builder** — A creational design pattern ***that lets you construct complex objects step by step***. The pattern allows you to produce different types and representations of an object using the same construction code.
4. **Singleton** — A creational design pattern ***that lets you ensure that a class has only one instance while providing a global access point to this instance.***
5. **Prototype** — A creational design pattern ***that lets you copy existing objects without making your code dependent on their classes.***

What is a Builder Design Pattern?

> *A **Builder Design Pattern** is a **Creational Design Pattern** that **lets you construct complex objects step by step.***

The pattern allows you to produce different types and representations of an object using the same construction code. It is similar to Factory and Abstract Factory Design Patterns.

Or We Can Say:

> *A **Builder Pattern** solves the issue with a large number of optional parameters and inconsistent states by providing a way to build the object step-by-step and provide a method that will actually return the final Object.*

Why Do We Need a Builder Design Pattern?

Builder Pattern is designed to create the objects using a nested public static class that has the same data fields as the outer class.

The **Builder Desing Pattern** was introduced to solve some of the problems with **Factory** and **Abstract Factory** design patterns when the object of the class contains a lot of field/data values.

Before we jump on Builder Pattern, let's see the issues with Factory and Abstract Factory Patterns for the scenarios where the object has a lot of field values.

1. ***Too Many arguments to pass from the client program*** to the Factory class can be error-prone because most of the time, the type of arguments are

the same and from the client side, it's hard to maintain the order of the argument.

2. Some of the parameters might be optional but in the Factory pattern, we are forced to send all the parameters and ***optional parameters need to send as `NULL`.***
3. ***If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that may be confusing.***

The Above Problems Can be Solved —

- When an Object Has a Large Number of Parameters by Just Provides a Constructor with the Required Parameters and Then Different Setter Methods to Set the Optional Parameters.
- But Do Remember, There Is a Problem With This Approach, That Is the Object State Will Be Inconsistent Unless All the Attributes Are Set Explicitly.

How to Implement Builder Design Pattern in Java?

If we follow the below-mentioned steps, we will have a step-by-step process to build the object and finally get the object —

1. Create a static nested class as a Builder class and then copy all the fields from the outer class to the Builder class. We should follow the naming convention. For example, if the class name is `Employee` then the builder class should be named as `EmployeeBuilder`.

2. The Builder class should have a public constructor with all the required fields as parameters.

3. The Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional field.

4. The final step is to provide a `build()` method in the builder class that will return the object needed by the client program. For this, we need to have a private constructor in the main class whose object needs to be constructed with the builder class as an argument.

Example:

We'll take a simple example to understand the builder design pattern working and behavior:

```java
public class Employee {

    private String name;
    private String company;
    private boolean hasCar;//optional
    private boolean hasBike;//optional

    private Employee(EmployeeBuilder employeeBuilder) {
        name = employeeBuilder.name;
        company = employeeBuilder.company;
        hasCar = employeeBuilder.hasCar;
        hasBike = employeeBuilder.hasBike;
    }

    public String getName() {
        return name;
    }
```

```
        public String getCompany() {
            return company;
        }

        public boolean isHasCar() {
            return hasCar;
        }

        public boolean isHasBike() {
            return hasBike;
        }

        public static class EmployeeBuilder {
            private String name;
            private String company;
            private boolean hasCar;//optional
            private boolean hasBike;//optional

            //constructor for required fields
            public EmployeeBuilder(String name, String company) {
                this.name = name;
                this.company = company;
            }

            //setter methods for optional fields
            public EmployeeBuilder setHasCar(boolean hasCar) {
                this.hasCar = hasCar;
                return this;
            }

            public EmployeeBuilder setHasBike(boolean hasBike) {
                this.hasBike = hasBike;
                return this;
            }

            //Build the Employee object
            public Employee build() {
                return new Employee(this);
            }
        }
    }

    class TestBuilder {
        public static void main(String[] args) {
            //Building the object of Employee thru the build() method provided in Em
            Employee employee = new Employee.EmployeeBuilder("Vikram", "ABC").setHas
        }
    }
```

Example of Builder Pattern: —

Many classes in Java use builder design patterns. For example `java.lang.StringBuilder` and `java.lang.StringBuffer` has used the builder pattern for object building.

Pros and Cons:

Pros:

- You can construct objects step-by-step, defer construction steps, or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- ***Single Responsibility Principle.*** You can isolate complex construction code from the business logic of the main class.

Cons:

- The overall complexity of the code increases since the pattern requires creating multiple new classes.

Relations with Other Patterns

- Many designs start by using the **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).
- **Builder** focuses on constructing complex objects step by step. **Abstract Factory** specializes in creating families of related objects. *Abstract Factory* returns the object immediately, whereas *Builder* lets you run some additional construction steps before fetching the object.
- You can use **Builder** when creating complex **Composite** trees because you can program its construction steps to work recursively.
- ***Abstract Factories*, *Builders*, and *Prototypes*** *can all be implemented as* ***Singletons***.

That's all for this article. Hope you liked it.

You Can Follow Vikram Gupta For Similar Content.

Builder Design · Design Pattern In Java · Low Level Design · Java Programming · Creational Design Pattern

Decorator Design Pattern

Knoldus Inc. · Follow
Published in Knoldus - Technical Insights · 3 min read · Jul 19, 2018

Hi everyone!

In this blog, we are going to discuss decorator design patterns with Scala.

Let's say I own a pizza outlet and we know that everyone has a very different taste and so there can be a various combination of toppings.

If I have n number of toppings, so i will have to create p(n) = 2 * p(n-1) + 1 Subclasses.

p(0) = 0

p(1) = 2 * p(1–1) + 1 = 1

p(2) = 2 * p(2–1) + 1 = 2 * p(1) + 1 = 2 * 1 + 1 = 3 p(3) = 2 * p2 + 1 = 2 * 3 + 1 = 7

p(4) = 2 * p3 + 1 = 2 * 7 + 1 = 15

So, if I have 3 toppings, the number of subclasses will be p(3) = 7, which is possible.

Wow!! 😲 My business is growing and now I want to expand it. So, I am going to add 2 more topping options for my valuable customers.

Now, when I have 5 toppings, the number of subclasses will be p(5) = 31. But wait!
It is really a very tedious task to be done 😞

So, what am I going to do now? 🤔

Am I going to drop the idea of expanding the business?

Nahh! I am going to use the decorator design pattern to solve this problem.

What is a design pattern?

Design patterns are best practices that a programmer can use to solve the problems that a programmer commonly faces when designing an application or system,
i.e. we can say, it is a general and reusable solution to a commonly occurring problem.

It is not a finished design which can be transformed directly into the source code but it is a description or template for how to solve a problem that can be used in many different situations.

Decorator design pattern:

Decorator design pattern is a structural design pattern.

Structural design patterns focus on Class and Object composition and decorator design pattern is about adding responsibilities to objects dynamically.

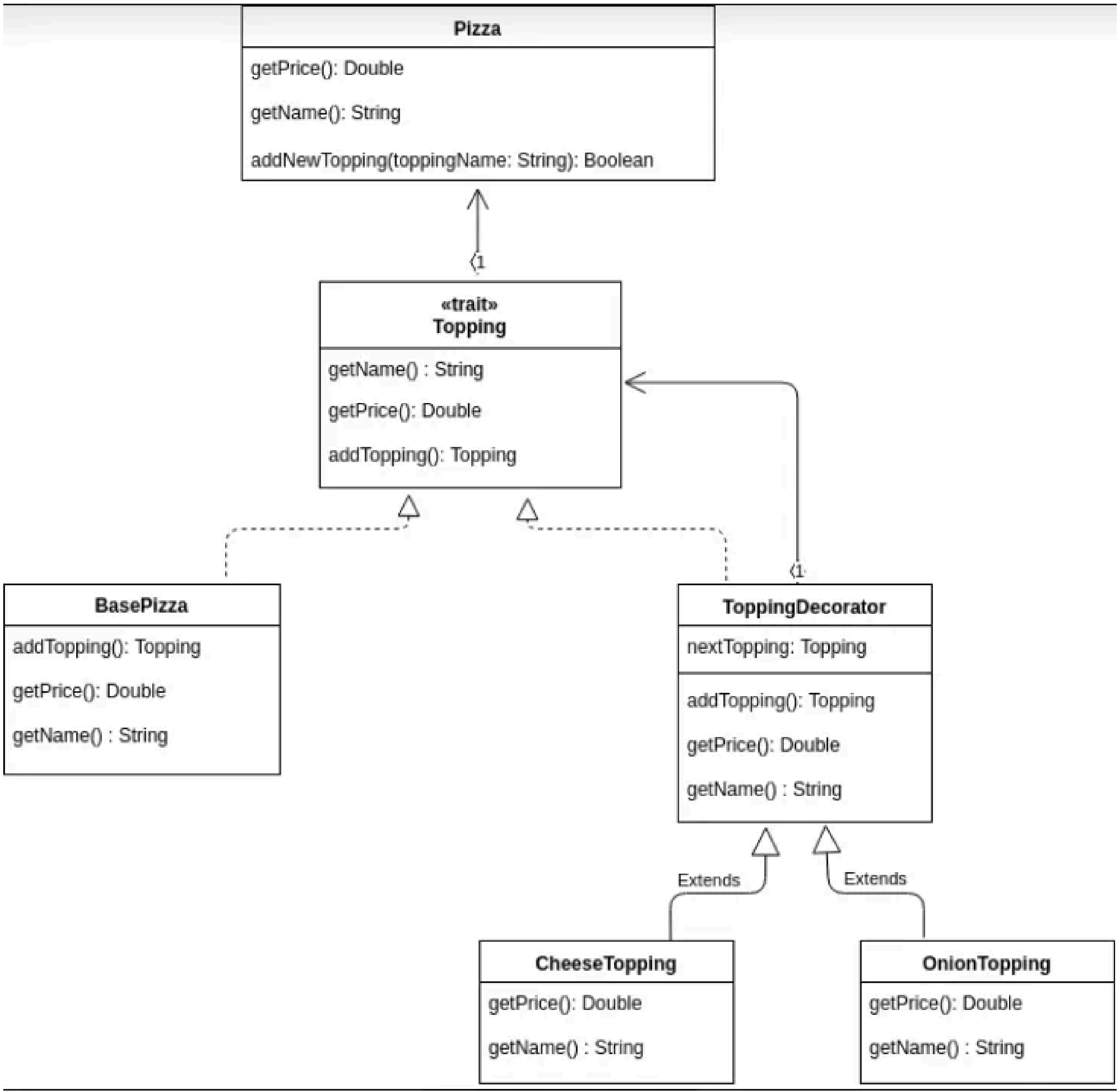
Decorator design pattern gives some additional responsibility to our base class.

This pattern is about creating a decorator class that can wrap original class and can provide additional functionality keeping class methods signature intact.

It is somewhat like the chain of responsibility pattern with the difference that in the chain of responsibility pattern, exactly one of the classes handles the request, while in decorator design pattern, all classes handle the request.

A design that uses Decorator often results in a system composed of lots of little objects that all look alike.

Following will be the UML diagram if we follow the Decorator design pattern to solve our problem :



Firstly, we have created **Topping** trait which is being implemented by classes **BasePizza** and **ToppingDecorator** and the **Pizza** class is composing it. **ToppingDecorator** is further being extended by classes **CheeseTopping** and **OnionTopping**.

```scala
class BasePizza extends Topping {
    def getName() : String = "Pizza"

    def getPrice() : Double = 77.0

    def addTopping() : Topping = this
}
```

BasePizza.scala hosted with ❤ by GitHub view raw

```scala
class CheeseTopping(override val topping : Topping) extends ToppingDecorator(topping) {
    override def getPrice() : Double = {
        super.getPrice() + 59.0
    }

    override def getName() : String = {
        val previous = super.getName()
        "Ocean Cheese " + previous
    }
}
```

CheeseTopping.scala hosted with ❤ by GitHub view raw

```scala
class OnionTopping(override val topping : Topping) extends ToppingDecorator(topping) {
    override def getPrice() : Double = {
        super.getPrice() + 39.0
    }

    override def getName() : String = {
        val previous = super.getName()
        "Sprinkled Onion " + previous
    }
}
```

OnionTopping.scala hosted with ❤ by GitHub view raw

```scala
class Pizza {
    var topping : Topping = new BasePizza

    def getPrice() : Double = {
        topping.getPrice()
    }

    def getName() : String = {
        topping.getName()
    }

    def addNewTopping(toppingName : String) : Boolean = {
        toppingName match
        {
            case "Onion" =>
            {
                this.topping = new OnionTopping(topping)
                true
            }
            case "Cheese" =>
            {
                this.topping = new CheeseTopping(topping)
                true
            }
            case _ =>
                println("Topping unavailable! Better luck next time! :(")
                false
        }
    }
}
```

Pizza.scala hosted with ❤ by GitHub view raw

```
object PizzaStore extends App {
    val pizza = new Pizza
    pizza.addNewTopping("Cheese")
    pizza.addNewTopping("Onion")
    println(s"You have ordered ${pizza.getName}")
    println(s"You have to pay Rupees ${pizza.getPrice}")
}
```

PizzaStore.scala hosted with ❤ by GitHub view raw

```
trait Topping {
    def getName() : String

    def getPrice() : Double

    def addTopping() : Topping
}
```

Topping.scala hosted with ❤ by GitHub view raw

```
class ToppingDecorator(val topping : Topping) extends Topping {
    var nextTopping : Topping = topping

    def getName() : String = nextTopping.getName()

    def getPrice() : Double = nextTopping.getPrice()

```

Composite design pattern with real example

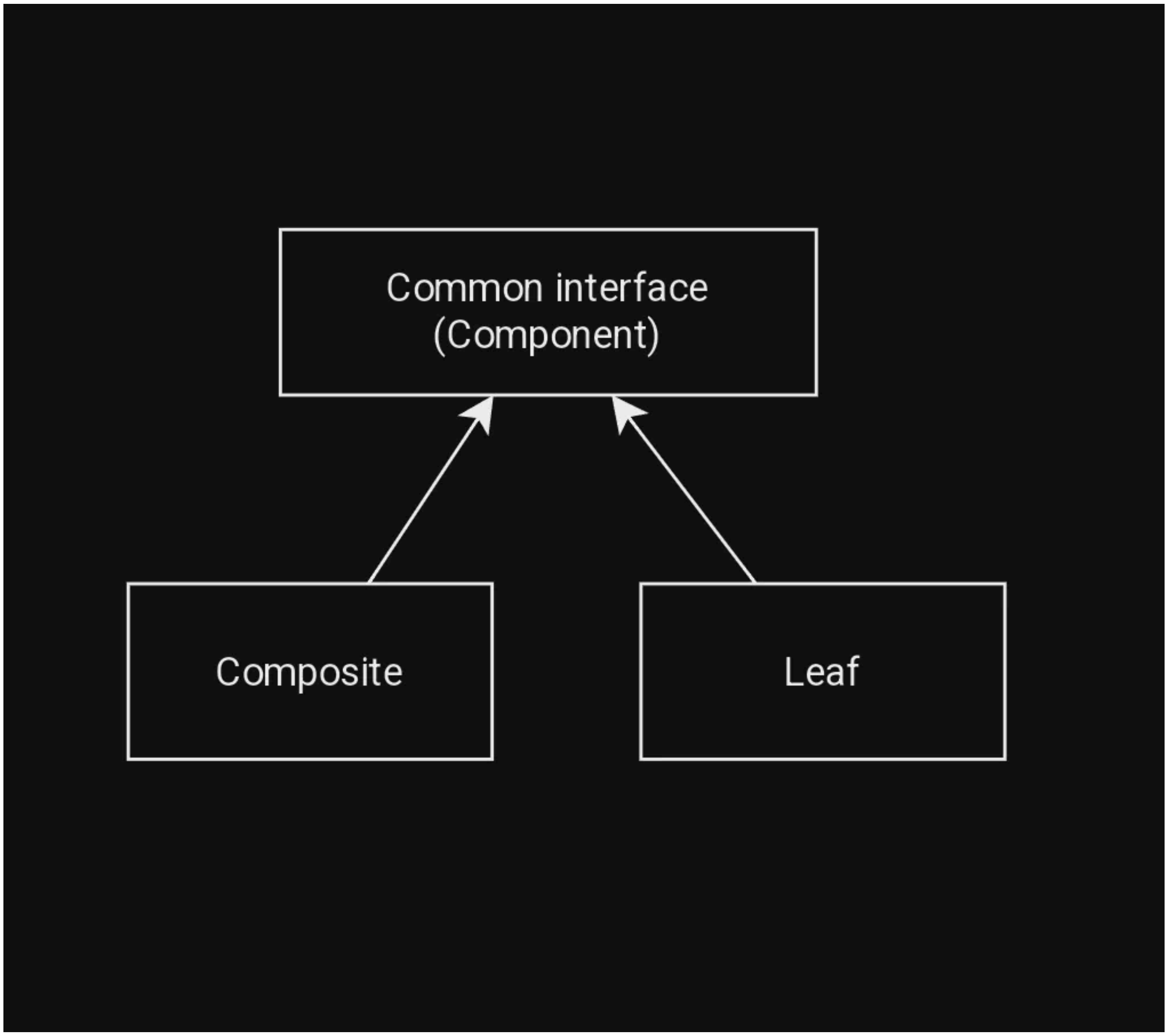
Willian Garbo · Follow
7 min read · Apr 2, 2024

composition to create large structure manageable) that helps you in the case where there is a hierarchical/recursive structure (i.e., tree structure) and common behaviors between entities. In the pattern, there is a composite that is an entity with children (including other composites recursively) and leaf that is the primitive entity without children. The composite pattern allow us to handle entities (composite or not) uniformly using a common interface (without has n if statements for each entity case), and add new entities without change existent code (open-closed principle from SOLID).

Structure:

- **A common interface (known as component)** with contract of common methods between composite and leaf entity (for both be used equally by client). Here, we can have some default implementation for some methods like add/remove child from composite.

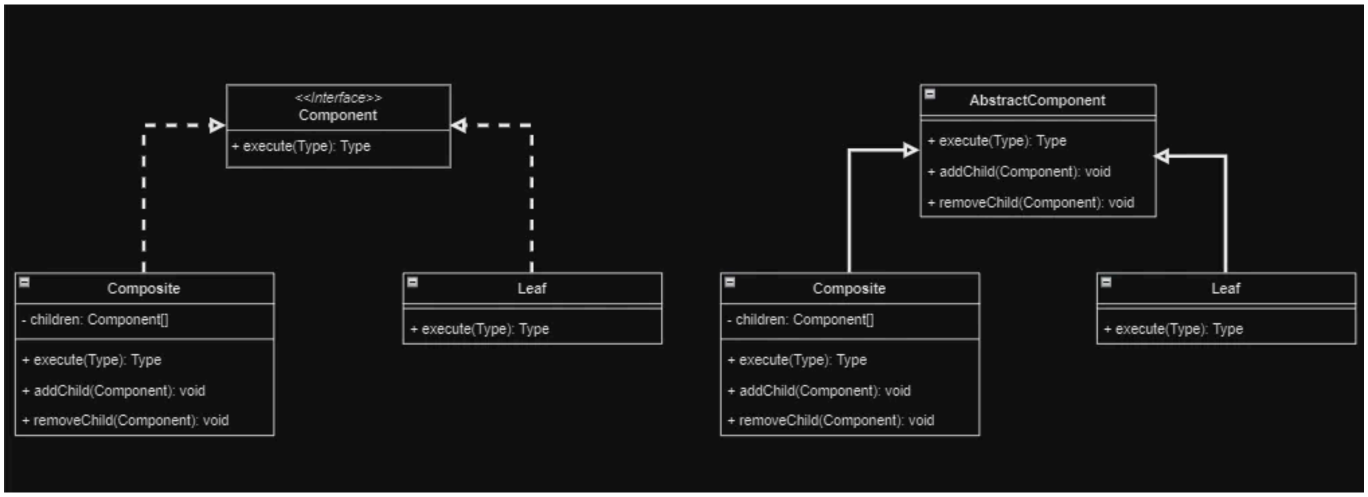
- **A composite entity** that implements the common interface and delegate operations to its children. Some actions can be done before or after delegate execution.
- **A leaf entity** that also implements the common interface and execute operation. This is the primitive entity.



Class diagram with common interface (component), composite and leaf.

With this structure the client can uses the component to interact with composites and leaves without know about specific class or implementation.

Implementation:



Class diagram with two ways that Composite Pattern can be represented.

Above there is a class diagram with two implementations ways of Composite Pattern. In the diagram on the left, we use composition (interface — **Component**) with an execute method, and two implementations of the interface. On the left, Composite class with children field, execute method and two additional methods addChild and removeChild to interact with declared field. On the right, just the leaf classe with execute method.

In the diagram on the right, we use inheritance by an abstract class (AbstractComponent) that class can define default behaviour by methods addChild and removeChild (methods required for composite entity), this because, in the leaf no makes sense override these methods. On the left of abstract class we find the Composite overriding execute, addChild and removeChild. And on the right leaf with execute method.

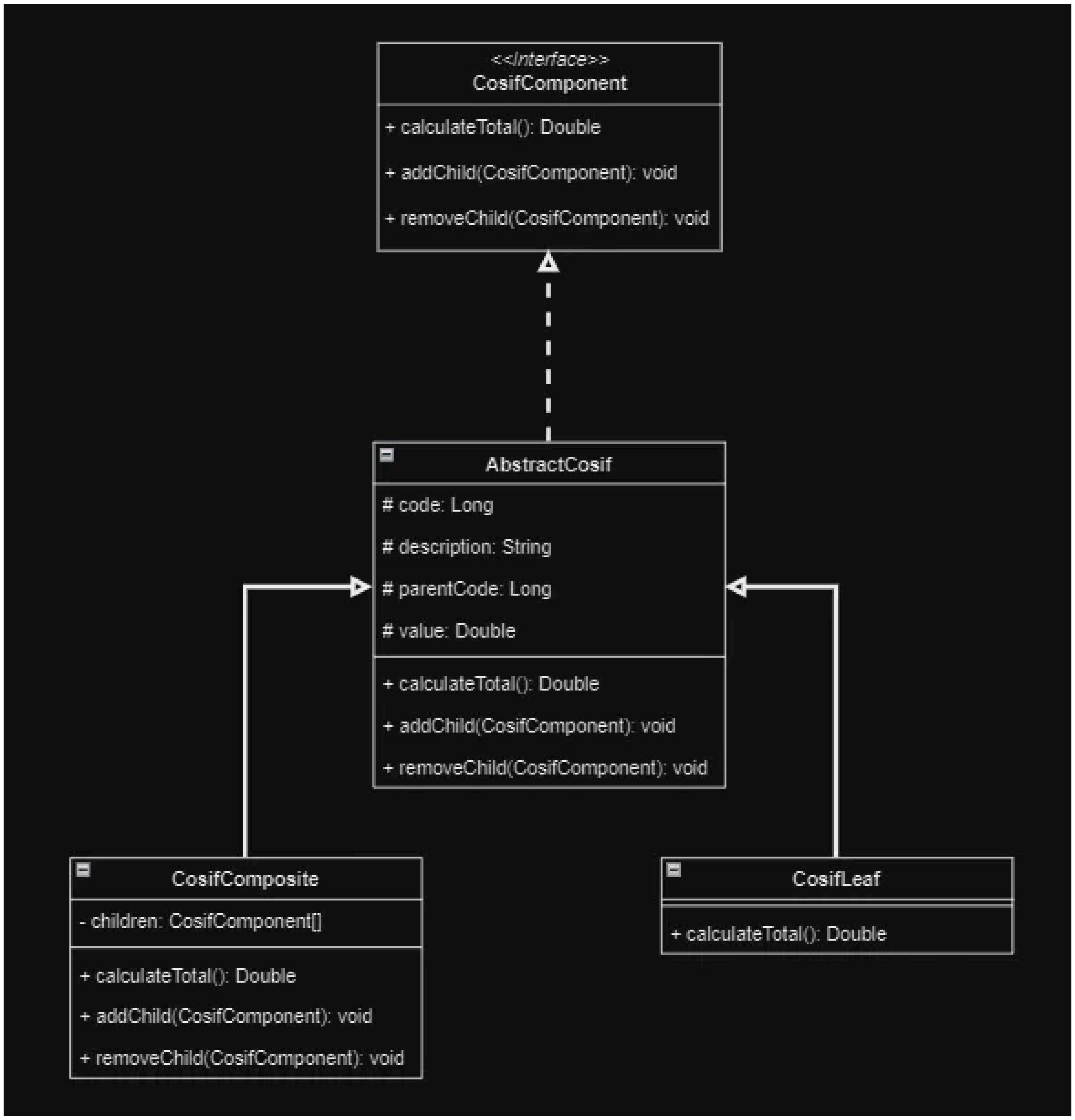
Example:

We will again use COSIF as an example (in a previous post, it was used with CTE).

The composites are cosifs with children, and leaves aren't. We will use code (Código), description (Descrição), parentCode (Conta Superior) fields from Cosif List. And we will add a additional field called value to our example.

Imagine that we need to calculate the total value of each entity (using the field value, added previously). With the composite pattern we can do it (instead of the composite, we could make some method to treat the cases depending of class, but here we will focus on structure and behavior in common objects). We call a method on the parent that delegates operations to the children. So each parent is responsible for sum all children´s values recursively and return it.

Note that we will use suffixes like Component, Composite and Leaf to make our example classes clear.


<<Interface>>
CosifComponent
+ calculateTotal(): Double
+ addChild(CosifComponent): void
+ removeChild(CosifComponent): void
AbstractCosif
# code: Long
# description: String
# parentCode: Long
# value: Double
+ calculateTotal(): Double
+ addChild(CosifComponent): void
+ removeChild(CosifComponent): void
CosifComposite
- children: CosifComponent[]
+ calculateTotal(): Double
+ addChild(CosifComponent): void
+ removeChild(CosifComponent): void
CosifLeaf
+ calculateTotal(): Double

Class diagram with our example. CosifComponent interface, AbstractCosif class, CosifComposite class and CosifLeaf class.

Above is our class diagram with Cosif classes using Composite Design Pattern. On the Top a CosifComponent was created, our interface with common methods (calculateTotal, addChild and removeChild). Below there is an abstract class with common cosif fields (present in all cosif classes) and this class implements CosifComponent and defines the default behavior for addChild and removeChild. On the bottom there are two concrete classes CosifComposite and CosifLeaf these extends our AbstractCosif.

Note that CosifComposite override default behavior of abstract.

Below we will see our classes diagram implementation usign Java. First we will see our interface CosifComponent.

```
package br.com.design.pattern.composite;

public interface CosifComponent
{
    void addChild(
        CosifComponent child );

    void removeChild(
        CosifComponent child );

    Double calculateTotal();
}
```

Next we will create an abstract class with some fields presents on Cosif. The abstract class implements our Component interface and define default actions for methods addChild and removeChild (throw Unsupported Operation as default to make sense for leaves).

```
package br.com.design.pattern.composite;

abstract class AbstractCosif
    implements
        CosifComponent
{
    private static final String UNSUPPORTED_OPERATION_ERROR_MESSAGE = "Method no

    protected Long code;
    protected String description;
    protected Long parentCode;
    protected Double value;
    public AbstractCosif(
        final Long code,
        final String description,
        final Long parentCode,
        final Double value )
    {
        this.code = code;
        this.description = description;
        this.parentCode = parentCode;
        this.value = value;
    }

    @Override
    public void addChild(
        final CosifComponent child )
    {
        throw new UnsupportedOperationException( UNSUPPORTED_OPERATION_ERROR_MES
    }

    @Override
    public void removeChild(
        final CosifComponent child )
    {
        throw new UnsupportedOperationException( UNSUPPORTED_OPERATION_ERROR_MES
    }

    public Long getCode() {
        return code;
    }

    public String getDescription() {
        return description;
    }

    public Long getParentCode() {
        return parentCode;
    }

    public Double getValue() {
        return value;
    }
}
```

Below we will create the leaf that only override the calculate total and just returns its value.

```
package br.com.design.pattern.composite;

public class CosifLeaf
    extends
        AbstractCosif
{
    public CosifLeaf(
        final Long code,
        final String description,
        final Long parentCode,
        final Double value )
    {
        super( code, description, parentCode, value );
    }

    @Override
    public Double calculateTotal()
    {
        return value;
    }
}
```

After that, we will create the composite entity with a list of children. Here we override the methods addChild and removeChild to make possible to client interact with children list. The calculateTotal method goes through all children and call the same method. Here the parent total is the sum of all children on hierarchy.

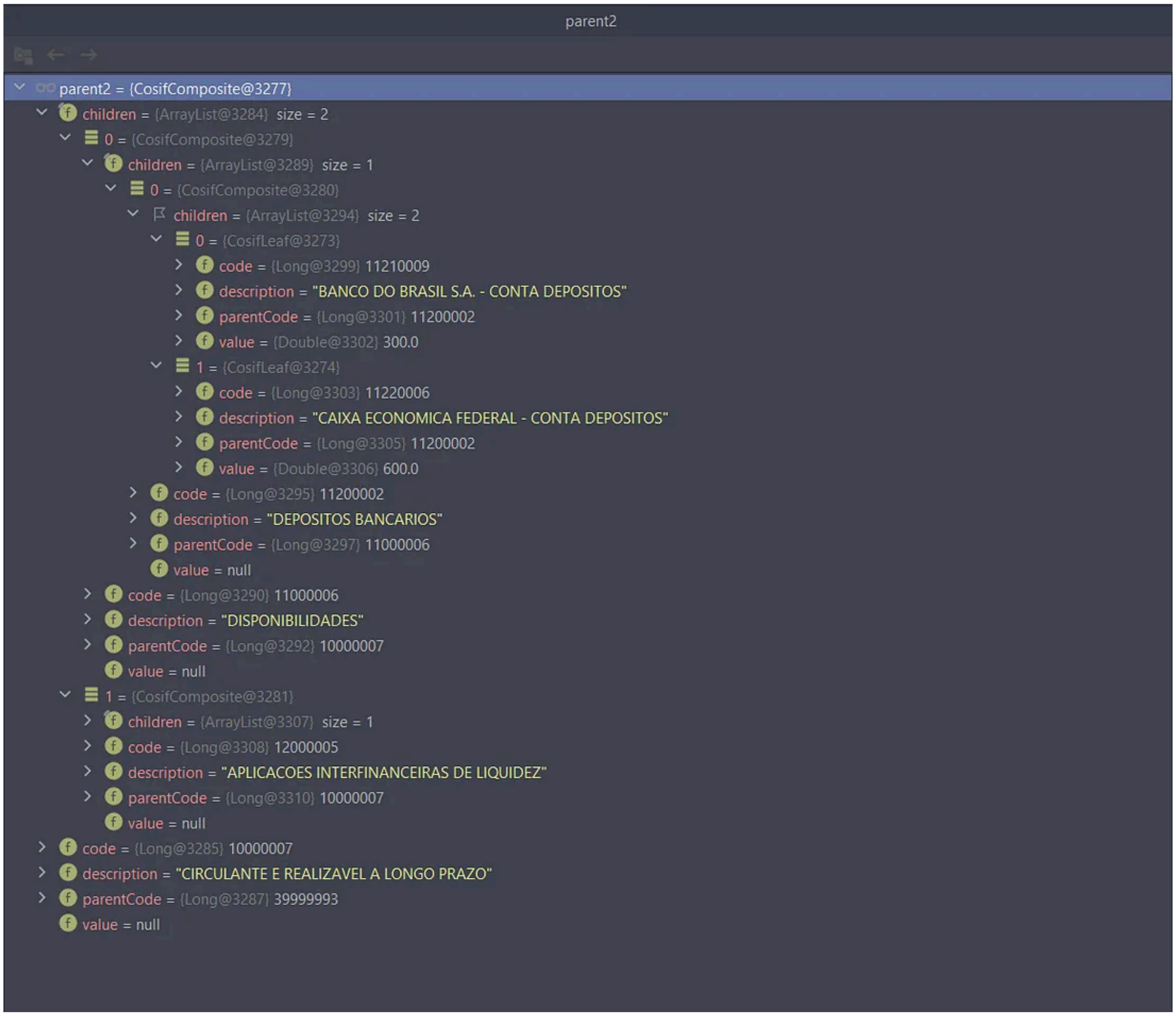
```
package br.com.design.pattern.composite;

import java.util.ArrayList;
import java.util.List;

public class CosifComposite
```

```
    extends
        AbstractCosif
{
    private final List<CosifComponent> children = new ArrayList<>();

    public CosifComposite(
        final Long code,
        final String description,
        final Long parentCode,
        final Double value )
    {
        super( code, description, parentCode, value );
    }

    @Override
    public void addChild(
        final CosifComponent child )
    {
        children.add( child );
    }

    @Override
    public void removeChild(
        final CosifComponent child )
    {
        children.remove( child );
    }

    /*//Equivalent
      Double sum = 0d;
      for( final CosifComponent child : children ) {
          sum += child.calculateTotal();
      }
      return sum;
    */
    @Override
    public Double calculateTotal()
    {
        return children.stream()
            .mapToDouble( CosifComponent::calculateTotal )
            .sum();
    }
}
```

Finally, we will use junit-jupiter to test some cases of calculateTotal (this class will be our client). Note that the test data was extracted from Cosif List.

```
package br.com.design.pattern.composite;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class CosifClientTest
{
    private final CosifComponent parent = new CosifComposite(
        39999993L, "TOTAL GERAL DO ATIVO",
        null, null );
    private final CosifComponent parent2 = new CosifComposite(
        10000007L, "CIRCULANTE E REALIZAVEL A LONGO PRAZO",
        39999993L, null );
    private final CosifComponent parent3 = new CosifComposite(
        11000006L, "DISPONIBILIDADES",
        10000007L, null );
    private final CosifComponent parent4 = new CosifComposite(
        11200002L, "DEPOSITOS BANCARIOS",
        11000006L, null );
    private final CosifComponent child = new CosifLeaf(
        11210009L, "BANCO DO BRASIL S.A. - CONTA DEPOSITOS",
        11200002L, 300. );
    private final CosifComponent child2 = new CosifLeaf(
        11220006L, "CAIXA ECONOMICA FEDERAL - CONTA DEPOSITOS",
        11200002L, 600. );

    private final CosifComponent parent5 = new CosifComposite(
        12000005L, "APLICACOES INTERFINANCEIRAS DE LIQUIDEZ",
        10000007L, null );
    private final CosifComponent parent6 = new CosifComposite(
        12100008L, "APLICACOES EM OPERACOES COMPROMISSADAS",
        12000005L, null );
    private final CosifComponent parent7 = new CosifComposite(
        12110005L, "REVENDAS A LIQUIDAR - POSICAO BANCADA",
        12100008L, null );
    private final CosifComponent child3 = new CosifLeaf(
        12110658L, "REVENDAS A LIQUIDAR - POSICAO BANCADA",
        12110005L, 999. );
    private final CosifComponent child4 = new CosifLeaf(
        12110359L, "LETRAS DE CAMBIO",
        12110005L, 76. );

    @BeforeEach
    private void setUp()
    {
        parent.addChild( parent2 );
        parent2.addChild( parent3 );
        parent3.addChild( parent4 );

        parent4.addChild( child );
        parent4.addChild( child2 );

        parent2.addChild( parent5 );

        parent5.addChild( parent6 );
        parent6.addChild( parent7 );

        parent7.addChild( child3 );
        parent7.addChild( child4 );
    }

    @Test
    @DisplayName( "Child value should be equal 300" )
    void shouldReturn300WhenCalculateTotalOfChild()
    {
        Assertions.assertEquals( Double.valueOf( 300d ), child.calculateTotal()
    }

    @Test
    @DisplayName( "Child 2 value should be equal 600" )
    void shouldReturn600WhenCalculateTotalOfChild2()
    {
        Assertions.assertEquals( Double.valueOf( 600d ), child2.calculateTotal()
    }

    @Test
    @DisplayName( "Child 3 value should be equal 999" )
    void shouldReturn999WhenCalculateTotalOfChild3()
    {
        Assertions.assertEquals( Double.valueOf( 999 ), child3.calculateTotal()
    }

    @Test
    @DisplayName( "Child 4 value should be equal 76" )
    void shouldReturn76WhenCalculateTotalOfChild4()
    {
        Assertions.assertEquals( Double.valueOf( 76 ), child4.calculateTotal() )
    }

    @Test
    @DisplayName( "Parent 4 value should be equal Child + Child2" )
    void shouldReturnSumOfChildPlusChild2WhenCalculateTotalOfParent4()
    {
        Assertions.assertEquals( child.calculateTotal() + child2.calculateTotal(
    }
    @Test
    @DisplayName( "Parent 7 value should be equal Child3 + Child4" )
    void shouldReturnSumOfChild3PlusChild4WhenCalculateTotalOfParent7()
    {
        Assertions.assertEquals( child3.calculateTotal() + child4.calculateTotal
    }

    @Test
    @DisplayName( "Parent 2 value should be equal all children sum" )
    void shouldReturnSumOfAllChildrenWhenCalculateTotalOfParent2()
    {
        Assertions.assertEquals( child.calculateTotal() + child2.calculateTotal(
            + child3.calculateTotal() + child4.calculateTotal(), parent2.calcula
    }

    @Test
    @DisplayName( "Parent value should be equal all children sum" )
    void shouldReturnSumOfAllChildrenWhenCalculateTotalOfParent()
    {
        final double childPlusChild2 = child.calculateTotal() + child2.calculate
        final double child3PlusChild4 = child3.calculateTotal() + child4.calcula
        final double allChildrenTotal = childPlusChild2 + child3PlusChild4;
        Assertions.assertEquals( allChildrenTotal, parent2.calculateTotal() );
        Assertions.assertEquals( allChildrenTotal, parent.calculateTotal() );
    }
}
```

In the image below, we will see our recursive structure. We have a parent2 entity from our test above. This entity has 2 children parent3 and parent5. The parent3 has parent4 as child and parent4 has 2 children (leaves) child and child2. The parent5 has parent6 as child and the parent6 has parent7 as child. The last entity parent7 has 2 children child3 and child4.


```
parent2
parent2 = {CosifComposite@3277}
  children = {ArrayList@3284}  size = 2
    0 = {CosifComposite@3279}
      children = {ArrayList@3289}  size = 1
        0 = {CosifComposite@3280}
          children = {ArrayList@3294}  size = 2
            0 = {CosifLeaf@3273}
              code = {Long@3299} 11210009
              description = "BANCO DO BRASIL S.A. - CONTA DEPOSITOS"
              parentCode = {Long@3301} 11200002
              value = {Double@3302} 300.0
            1 = {CosifLeaf@3274}
              code = {Long@3303} 11220006
              description = "CAIXA ECONOMICA FEDERAL - CONTA DEPOSITOS"
              parentCode = {Long@3305} 11200002
              value = {Double@3306} 600.0
          code = {Long@3295} 11200002
          description = "DEPOSITOS BANCARIOS"
          parentCode = {Long@3297} 11000006
          value = null
      code = {Long@3290} 11000006
      description = "DISPONIBILIDADES"
      parentCode = {Long@3292} 10000007
      value = null
    1 = {CosifComposite@3281}
      children = {ArrayList@3307}  size = 1
      code = {Long@3308} 12000005
      description = "APLICACOES INTERFINANCEIRAS DE LIQUIDEZ"
      parentCode = {Long@3310} 10000007
      value = null
  code = {Long@3285} 10000007
  description = "CIRCULANTE E REALIZAVEL A LONGO PRAZO"
  parentCode = {Long@3287} 39999993
  value = null
```

Parent2 tree. Imagem from intellij.

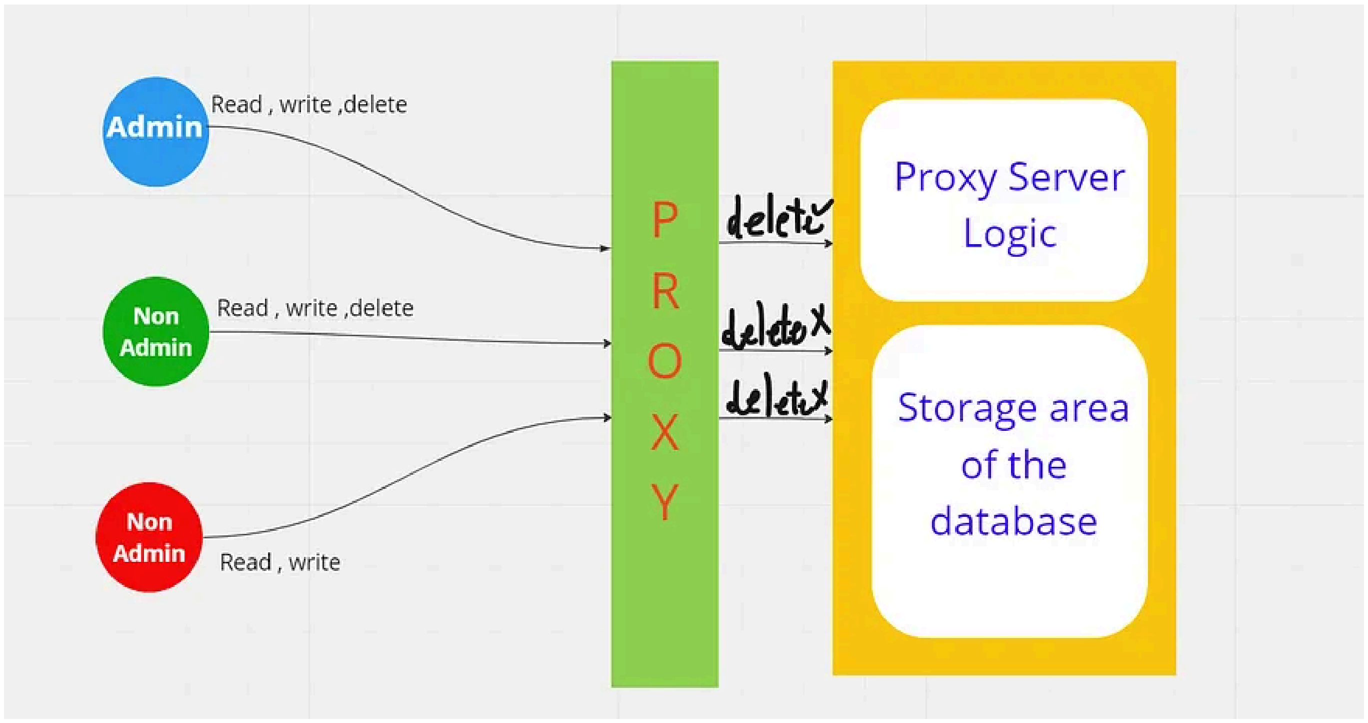
(Dis)Advantages:

- Open-close principle — *advantage*;
- Common entity contract (client handles entities without distinction) — *advantage*;
- Sometimes, the contract goes over Interface segregation/Liskov substitution principle (SOLID— I and L) because some methods cannot fit well on leaf (in own example addChild and removeChild) — *disadvantage*;
- It’s hard to debug when compositions are too large — *disadvantage*;

What is Proxy Design Pattern ?

> *The Proxy Design Pattern is a structural design pattern that provides a surrogate or placeholder for another object to control access to it.*

To understand it better let’s consider an example -: Suppose you are a service provider and you have a database that contains some useful information and this database can be modified by you only . Now you gave your service to your client and that client can alter your database , which is not acceptable . So to avoid this you will introduce a proxy class that client can have access to and any command given by client is controlled by this proxy class hence making your database more secure.



Components of Proxy Design Pattern

The main components used in Proxy Design Pattern are -:

1. **Real Subject** -: The `Real Subject` is the actual object that the `Proxy` represents. It contains the real implementation of the business logic or the resource that the client code wants to access.
2. **Proxy** -: The `Proxy` acts as a surrogate or placeholder for the `Real Subject`. It controls access to the real object and may provide additional functionality .
3. **Subject -:** The `Subject` is an interface or an abstract class that defines the common interface shared by the `Real Subject` and `Proxy` classes.

Implementation of Proxy Design Pattern

To illustrate Proxy Pattern let's consider a simple example of an **Database Executer interface** which can act as **admin or non admin** and can make request to the **Database Proxy** which in turn can be served by **real database** if required .

Subject Class -:

```
public interface DataBaseExecutor
{
    public abstract void executeQuery(String Type);
}
```

This `subject` interface provides all the necessary methods to be executed by `Real Subject` and `proxy` classes .

Real Subject Class -:

```
import java.util.Objects;

public class DataBase
{
     public DataBase()
     {

     }
     void execute(String QueryType,String id)
     {
         System.out.println("executing the query : " + QueryType);
     }
}
```

Proxy Class -:

```
import java.util.Objects;

public class DataBaseProxy implements DataBaseExecutor
{
    String id;

    DataBase db;
    private DataBaseProxy()
    {

    }
    public DataBaseProxy(String id)
    {
        this.id = id;
        db = new DataBase();        //connection
    }

    @Override
    public void executeQuery(String QueryType)
    {
        if(QueryType.equals("DELETE") && !Objects.equals(this.id, "ADMIN"))
        {
            System.out.println("cannot execute the query , admin permission is n
            return;
        }
         db.execute(QueryType,this.id);
    }
}
```

Client Class -:

```
public class Main
{
    public static void main(String[] args)
    {
        DataBaseExecutor emp1 = new DataBaseProxy("NON ADMIN");
        emp1.executeQuery("READ");
        emp1.executeQuery("WRITE");
        emp1.executeQuery("DELETE");
```

```
            DataBaseExecutor emp2 = new DataBaseProxy("ADMIN");
            emp2.executeQuery("READ");
            emp2.executeQuery("WRITE");
            emp2.executeQuery("DELETE");

        }
    }
```

In client class , client can give desired query to proxy and according to client type query is being executed .

Advantages

The Proxy Design Pattern offers several advantages, and here are some brief points highlighting its benefits:

Controlled Access:

- Proxies control access to the real subject, allowing for additional checks or restrictions.

Lazy Initialization:

- Proxies can delay the creation and initialization of the real subject until it is actually needed, improving performance.

Security:

- Proxies can provide an additional layer of security by implementing authentication or authorization checks before allowing access to the real subject.

Remote Access:

- Remote proxies can represent objects that are in different address spaces, enabling communication between distributed components.

Logging and Monitoring:

- Proxies can be used to log and monitor requests, providing insights into the behavior of the real subject.

Caching:

- Proxies can implement caching mechanisms to store and reuse the results of expensive operations, improving performance.

Load Balancing:

- Proxies can be used for load balancing by distributing requests among multiple real subjects.

Simplified Interface:

- Proxies can provide a simplified or restricted interface to the real subject, exposing only the necessary methods.

Reduced Resource Usage:

- Virtual proxies can be used to represent large or resource-intensive objects, loading them into memory only when required.

Transparent Functionality Extension:

- Proxies can transparently add functionality, such as logging, without modifying the code of the real subject.

Concurrency Control:

- Proxies can implement concurrency control mechanisms, such as locking, to manage access to shared resources.

Decoupling:

- Proxies promote a decoupling between the client and the real subject, making it easier to introduce changes or alternative implementations.

So this was everything you need to know for Proxy Design Pattern . Hope you liked it . 😄

Proxy Design Pattern | Design Patterns | System Design Interview | Java | Low Level Design

Written by Neha Gupta

92 Followers

Follow

More from Neha Gupta

Chain of responsibility design pattern(COP)

Prakash Sharma · Follow
4 min read · May 18, 2023

This is one of the important design patterns being used in software development and today we will see a simple example that demonstrates the usage and reasoning behind it.

Problem statement: Implement a web-server request handler that preprocesses the request for authentication, logging, compression, etc. The server should be open to extension and closed for modification, which means tomorrow if we want to add some other actions like encryption, priority, etc then we should be able to do it without modifying the server code.

If we read the problem statement closely it is talking about creating a pipeline of actions, there are two things which possible here, first — we have a limited set of actions that will **never change,** and second case where we know that in the future these actions will be changed[maybe the order] or some new actions will be added.

We know that in software development there is nothing constant, only "**change is constant**", and we should always write code that is extensible in nature. so let's see how can we implement this in an extensible way 😎.

Understanding the Chain of Responsibility Design Pattern: The Chain of Responsibility design pattern is a behavioral pattern that enables us to build a chain of objects, each having the ability to process a request and pass it to the next object in the chain. This pattern promotes loose coupling and encapsulates request-handling logic within individual components.

Implementing the Web Server Request Handler: To achieve an extensible web server request handler, we'll design a system that processes requests in a chain-like manner, with each handler responsible for a specific action. Let's take a closer look at the implementation steps:

Step 1: Define the RequestHandler Interface: Create an abstract class, `Handler`, that declares the common methods for handling requests. This interface should include a method for setting the successor handler and another method for handling the request itself.

```java
package com.company.cop;

public abstract class Handler {
    private Handler next;

    public Handler(Handler next) {
        this.next = next;
    }

    public void handle(HttpRequest request){
        if (doHandle(request)){
            return;
        }

        if (next != null){
            next.handle(request);
        }
    }

    abstract boolean doHandle(HttpRequest request);
}
```

Handler.java hosted with ❤ by GitHub view raw

Step 2: Implement the Concrete Request Handlers: Create concrete request handlers by implementing the `Handler` interface. Each handler should contain the logic for a specific action such as authentication, logging, compression, encryption, priority, and so on. Handlers can be as simple or complex as required, depending on the desired functionality.

```java
package com.company.cop;

public class Authenticator extends Handler {
    public Authenticator(Handler next) {
        super(next);
    }

    private boolean authenticate(HttpRequest request){
        System.out.println("Authentication");
        return request.getUsername().equals("admin") && request.getPassword().equals("password")
    }

    @Override
    boolean doHandle(HttpRequest request) {
        return !authenticate(request);
    }
}
```

Authenticator.java hosted with ❤ by GitHub view raw

```java
package com.company.cop;

public class Compressor extends Handler{
    public Compressor(Handler next) {
        super(next);
    }

    @Override
    boolean doHandle(HttpRequest request) {
        System.out.println("Compressing");
        return false;
    }
}
```

Compressor.java hosted with ❤ by GitHub view raw

```java
package com.company.cop;

public class Logger extends Handler {
    public Logger(Handler next) {
        super(next);
    }

    @Override
    boolean doHandle(HttpRequest request) {
        System.out.println("Logging");
        return false;
    }
}
```

Logger.java hosted with ❤ by GitHub view raw

The request object looks like this —

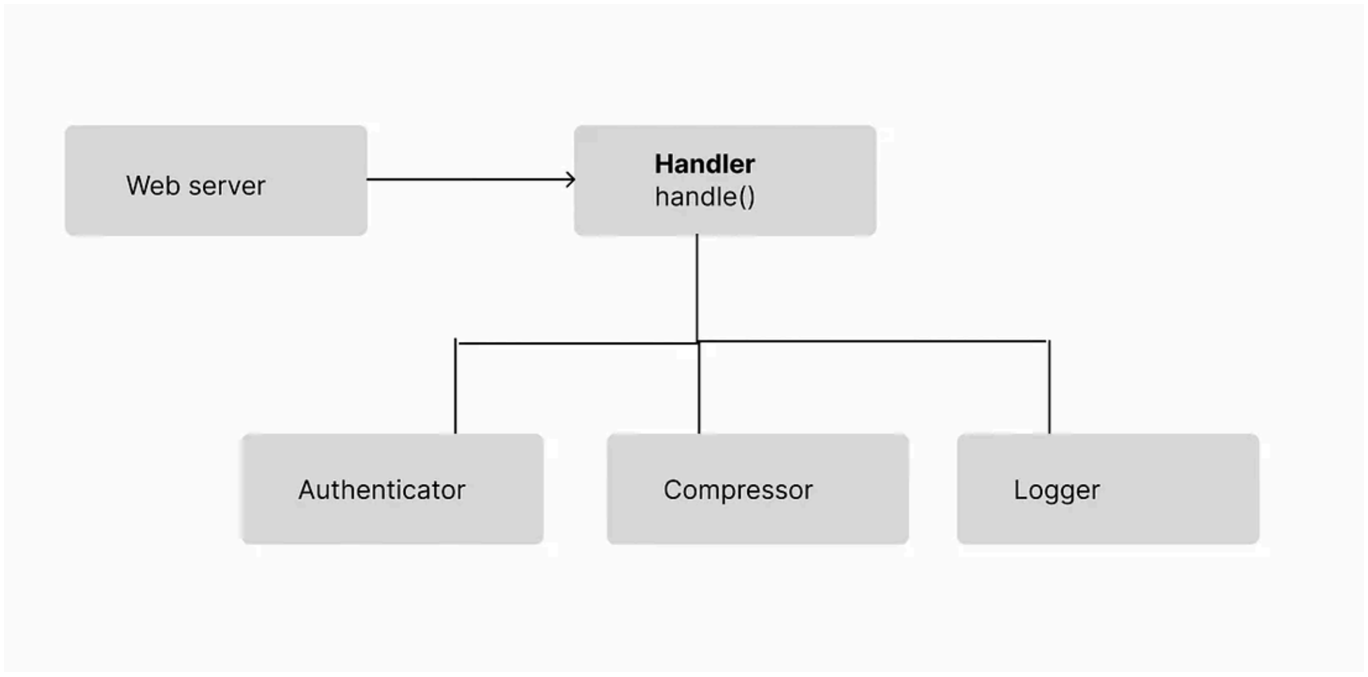
```java
package com.company.cop;

public class HttpRequest {

    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    public HttpRequest(String username, String password) {
        this.username = username;
        this.password = password;
    }


}
```

HttpRequest.java hosted with ❤ by GitHub view raw

Step 3: Process the Request: Create a request processor that accepts incoming requests and passes them through the chain of responsibility. The processor will invoke the first handler, which will in turn pass the request to the next handler in the chain until the end of the chain is reached. Each handler can perform its designated action on the request before passing it along to the next handler.

```java
package com.company.cop;

public class WebServer {

    private Handler handler;

    public WebServer(Handler handler) {
        this.handler = handler;
    }

    public void handle(HttpRequest request){
        handler.handle(request);
    }
}
```

WebServer.java hosted with ❤ by GitHub view raw

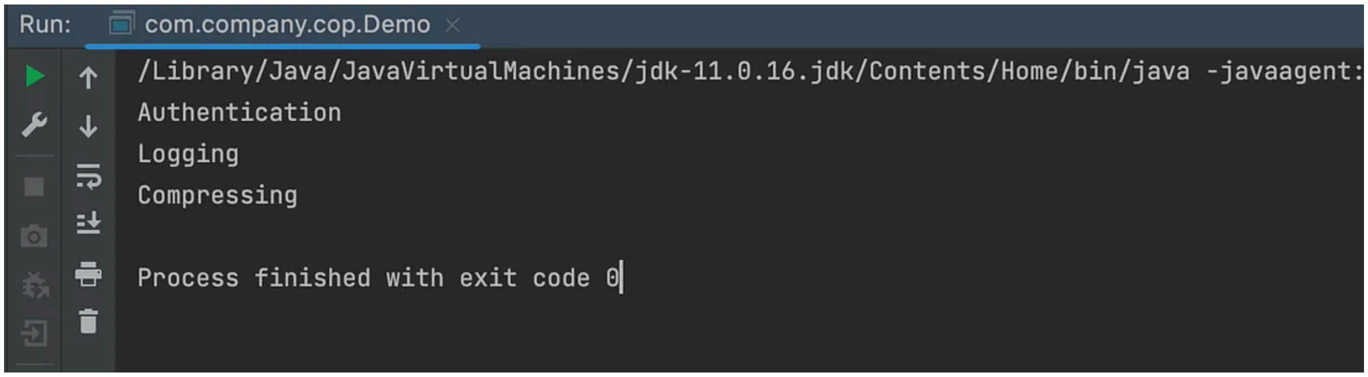


And finally demo of the above implementation —

```java
package com.company.cop;

public class Demo {
    public static void main(String[] args) {
        Authenticator authenticator = new Authenticator(new Logger(new Compressor(null)));
        WebServer server = new WebServer(authenticator);
        HttpRequest request = new HttpRequest("admin", "password");
        server.handle(request);
    }
}
```

Demo.java hosted with ❤ by GitHub view raw

Output:



```
/Library/Java/JavaVirtualMachines/jdk-11.0.16.jdk/Contents/Home/bin/java -javaagent:
Authentication
Logging
Compressing

Process finished with exit code 0
```

Benefits of the Chain of Responsibility Approach: By using the Chain of Responsibility design pattern, we achieve several advantages:

1. Extensibility: The server becomes open to extension, allowing us to add new actions such as encryption, priority handling, or any other functionality without modifying the existing server code. Each new action can be implemented as a separate handler and seamlessly integrated into the existing chain.

2. Modular and Reusable Components: Each handler encapsulates a specific action, making the codebase modular and promoting code reusability. Handlers can be easily rearranged or replaced, enabling flexible customization of the request handling process.
3. Maintainability and Separation of Concerns: The chain of responsibility separates the concerns of different actions. Each handler focuses on a specific task, reducing the complexity of individual components and facilitating easier maintenance. Modifications or updates required for a particular action can be made in its respective handler, minimizing the impact on other parts of the system.

Conclusion: In this blog post, we explored how the Chain of Responsibility design pattern can be used to implement an extensible web server request handler. By leveraging the pattern, we achieved a server that is open to extension and closed for modification, allowing easy integration of new actions without altering the core server code. This approach promotes modular and reusable components, enhances maintainability, and enables separation of concerns. With the Chain of Responsibility pattern, we can build powerful and adaptable web servers that meet the evolving needs of our applications.

Implementing the Chain of Responsibility pattern in your web server request handler not only facilitates extensibility but also lays a foundation for a scalable and robust system.

Hope you liked the post, and do not forget to clap and subscribe for more such posts.

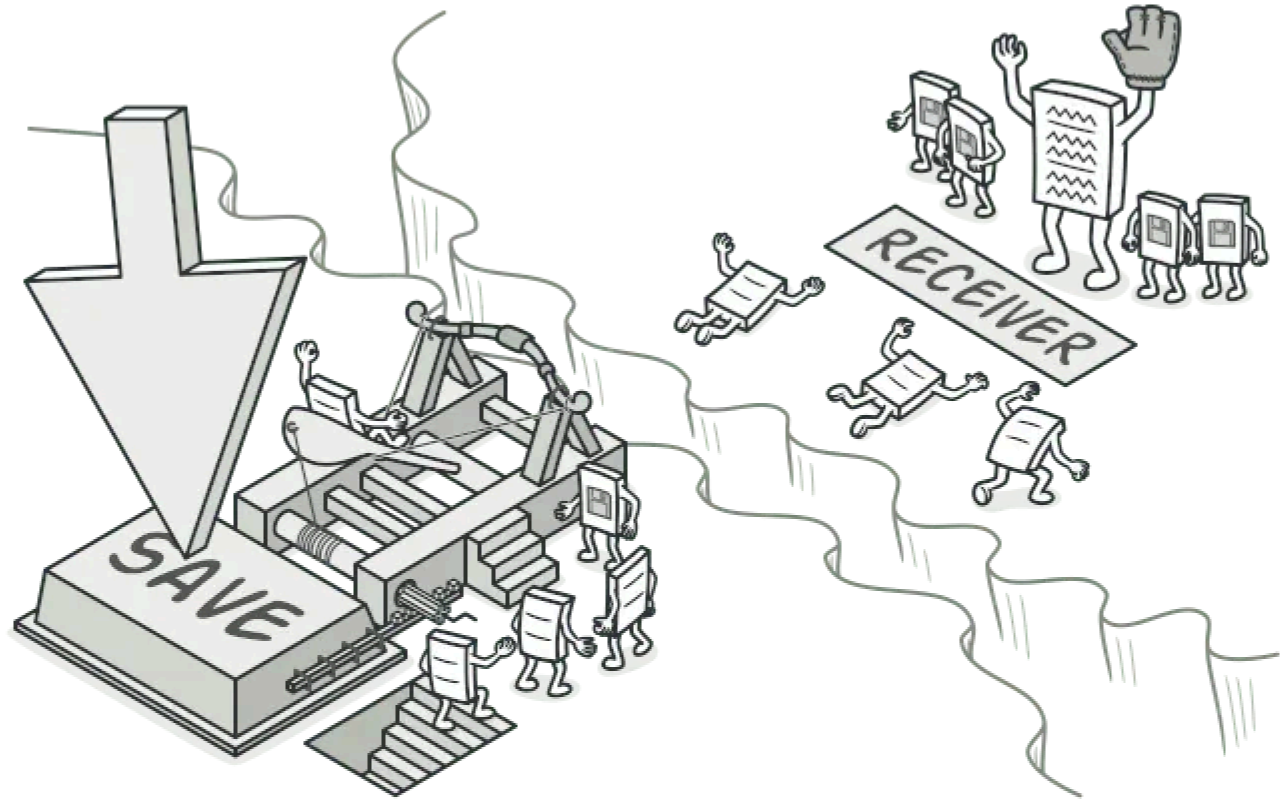
You can checkout my previous posts

Simplifying Command Execution with the Command Design Pattern in Java

Rajeshvelmani · Follow
4 min read · Jul 17, 2023

4 1

In software development, the Command design pattern provides a way to encapsulate a request as an object, thereby decoupling the sender and receiver of the request. This pattern allows for the parameterization of clients with different requests, queuing or logging requests, and supporting undoable operations. In this blog, we will explore the Command design pattern, understand its implementation in Java, and provide a practical example to illustrate its usage.



https://refactoring.guru/design-patterns/command

Understanding the Command Design Pattern:

The Command design pattern falls under the behavioral design pattern category. It encapsulates a request as an object, encapsulating all the necessary information to perform an action, including the receiver object and the specific method to invoke. The pattern promotes loose coupling between the sender and receiver, allowing for flexibility in executing commands and supporting various operations, such as undo and redo.

Key Features of the Command Design Pattern:

1. **Command:** The command interface or abstract class defines the common methods that encapsulate different requests. It typically declares an `execute()` method to perform the requested action.
2. **Concrete Commands:** Concrete command classes implement the command interface or extend the command abstract class. Each concrete

command encapsulates a specific request by holding a reference to the receiver object and invoking the appropriate method.

3. **Receiver:** The receiver is the object that performs the actual operations associated with a command. It knows how to execute the request and carries out the specific actions.

4. **Invoker:** The invoker class is responsible for invoking the command and directing it to the appropriate receiver. It maintains a reference to the command and triggers its execution.

Implementing the Command Design Pattern in Java:

Let's demonstrate the implementation of the Command design pattern using Java code. We'll create an example of a simple remote control that allows commands to be executed on different devices, such as turning on/off a television or adjusting the volume.

```
// Command
interface Command {
    void execute();
}
// Concrete Commands
class TurnOnCommand implements Command {
    private Television television;
    public TurnOnCommand(Television television) {
        this.television = television;
    }
    @Override
    public void execute() {
        television.turnOn();
    }
}
class TurnOffCommand implements Command {
    private Television television;
    public TurnOffCommand(Television television) {
        this.television = television;
    }
    @Override
```

```java
    public void execute() {
        television.turnOff();
    }
}
class VolumeUpCommand implements Command {
    private Television television;
    public VolumeUpCommand(Television television) {
        this.television = television;
    }
    @Override
    public void execute() {
        television.volumeUp();
    }
}
// Receiver
class Television {
    public void turnOn() {
        System.out.println("Television turned on.");
    }
    public void turnOff() {
        System.out.println("Television turned off.");
    }
    public void volumeUp() {
        System.out.println("Volume up.");
    }
}
// Invoker
class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void pressButton() {
        command.execute();
    }
}
```

> *The `Command` interface defines the command interface. It declares the `execute()` method to encapsulate the requested action.*
>
> *The `TurnOnCommand`, `TurnOffCommand`, and `VolumeUpCommand` classes are concrete command implementations. They implement the `Command` interface and*

> *encapsulate specific requests by holding a reference to the receiver object and invoking the appropriate methods.*
>
> *The `Television` class represents the receiver. It knows how to execute the requested actions such as turning on/off or adjusting the volume.*
>
> *The `RemoteControl` class acts as the invoker. It maintains a reference to the command and triggers its execution by calling the `execute()` method.*

Using the Command: To use the Command pattern in the remote control example, clients can create instances of the concrete commands, the receiver (television), and the invoker (remote control).

```
public class Application {
    public static void main(String[] args) {
        Television television = new Television();
        Command turnOnCommand = new TurnOnCommand(television);
        Command turnOffCommand = new TurnOffCommand(television);
        Command volumeUpCommand = new VolumeUpCommand(television);
        RemoteControl remoteControl = new RemoteControl();
        remoteControl.setCommand(turnOnCommand);
        remoteControl.pressButton(); // Turns on the television
        remoteControl.setCommand(volumeUpCommand);
        remoteControl.pressButton(); // Increases the volume
        remoteControl.setCommand(turnOffCommand);
        remoteControl.pressButton(); // Turns off the television
    }
}
```

In the above example, we create instances of the `TurnOnCommand`, `TurnOffCommand`, and `VolumeUpCommand` concrete commands. We also create an

Open in app ↗

invoke the command by calling the `pressButton()` method.

Advantages of the Command Design Pattern:

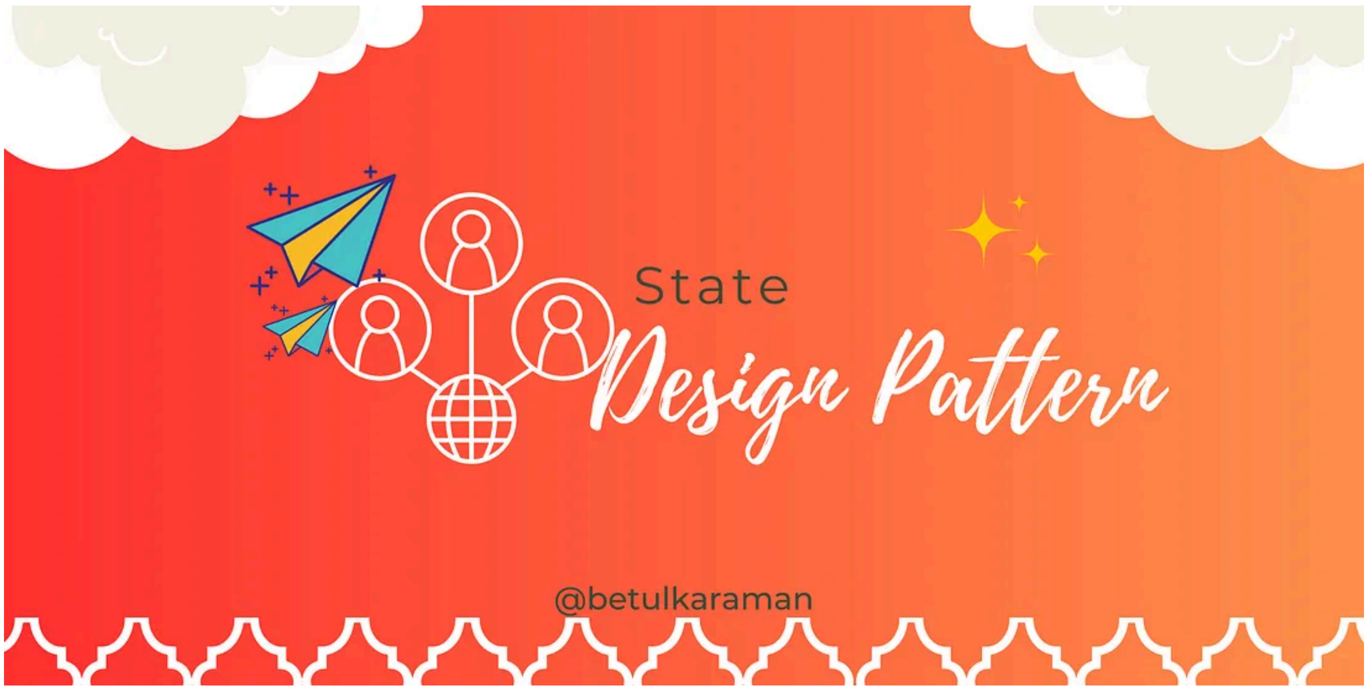
1. **Decoupling of components:** The Command pattern decouples the sender and receiver of a request by encapsulating the requestas an object. The sender doesn't need to know the specific receiver or how the request is executed, promoting loose coupling and enhancing code maintainability.
2. **Flexibility and extensibility:** New commands can be added easily without modifying existing code. The pattern allows for the dynamic configuration of commands at runtime and supports various operations, such as undo and redo.
3. **Separation of concerns:** Each concrete command encapsulates a specific request, promoting a separation of concerns and making the code more modular and maintainable.
4. **Logging and queuing of requests:** The Command pattern facilitates logging or queuing of requests, allowing for the implementation of features like command history or transaction management.

Conclusion:

The Command design pattern provides a clean and flexible way to encapsulate requests as objects, promoting loose coupling and enhancing code maintainability. In Java, the Command pattern can be applied in various scenarios where requests need to be decoupled from the sender and receiver, supporting flexibility, extensibility, and advanced command execution features. By utilizing the Command pattern effectively, you can achieve code reusability, modularity, and easier management of complex operations, ultimately enhancing the overall flexibility and maintainability of your codebase.

State Management with State Design Pattern

betulkaraman · Follow
6 min read · Jan 12, 2024



In software development processes, various design patterns are employed to make complex systems more manageable and flexible. These patterns provide software engineers with more effective solutions to tackle problems, ensuring that the code is readable, sustainable, and extensible.

One of these patterns is the "State" pattern, which is based on the idea of representing object states with a series of separate objects. Thus, the behavior of an object can change as its state changes, while the object itself remains unchanged. The State pattern offers an effective way, especially for transitioning between states and managing these transitions effortlessly.

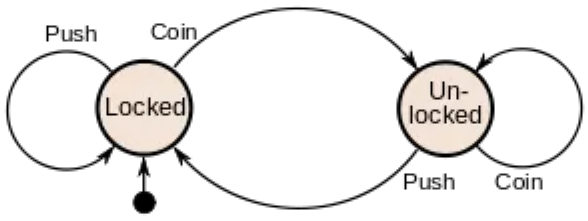
This article encompasses fundamental concepts, examples, and implementation guidelines to help you understand and apply the State pattern. Once you grasp how the State pattern works, you'll possess a tool that facilitates writing more modular, maintainable, and extensible code in your software projects.

If you're ready, let's begin!

The State design pattern is a software design pattern used to represent the internal state of an object and alter its behavior based on this state. This pattern models different states of an object as separate objects, and as the object's state changes, the corresponding object representing that state is replaced. Consequently, the object's behavior can be dynamically altered based on its state.

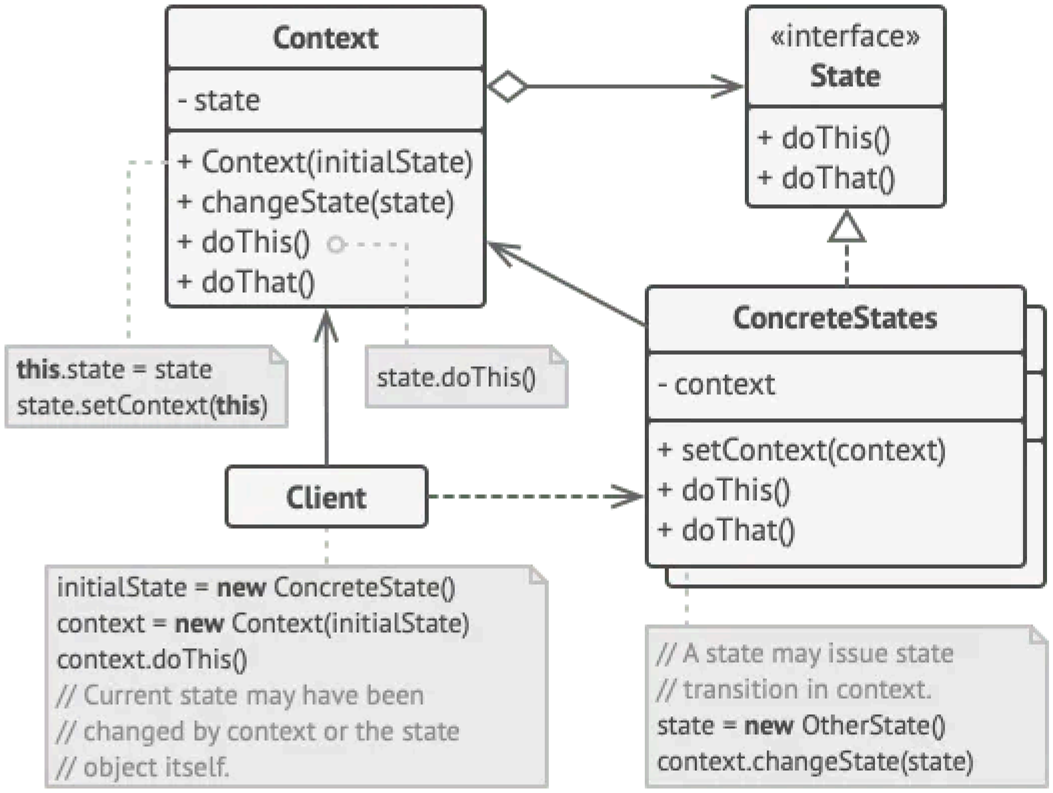
The State pattern is particularly employed to manage transitions between states and represent the internal state of an object as a series of independent classes. This facilitates code to be more modular, readable, and easier to maintain.

The State pattern is closely related to the concept of a Finite-State Machine (FSM). How so? A Finite-State Machine, or FSM, is a mathematical model that represents the states in a system and the transitions between these states. FSM is defined by a set of states, a set of transitions, and an initial state. Each state represents a specific condition in the system, and transitions denote shifts from one state to another when a certain event or condition occurs in the system.



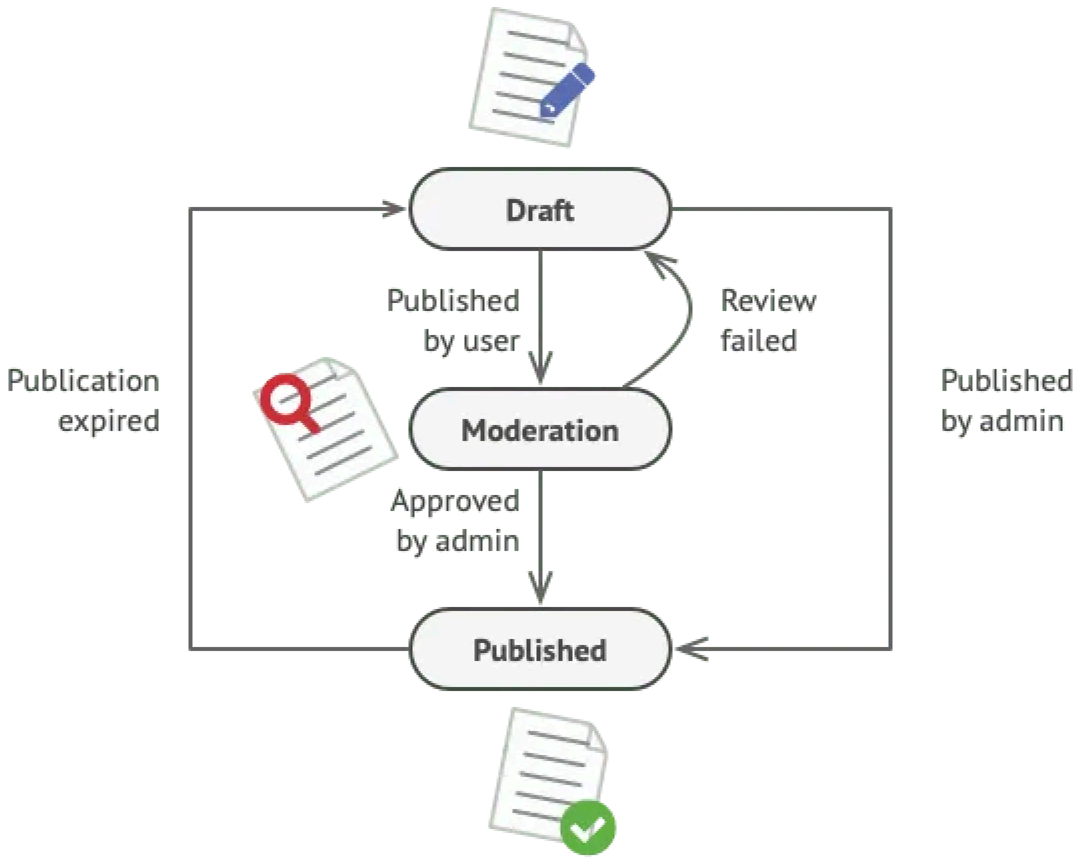
Finite State Machine

Finite State Machine (FSM), especially used to clearly model different states and transitions between them in a system. An FSM is utilized to understand, design, and analyze specific system behavior. The State design pattern assists in implementing this FSM concept because the object's state and state changes are modeled with the State pattern, providing a structure similar to the fundamental principles of FSM.



State Design Pattern Diagram

Let's consider a scenario where you have a class named `Document` in your software project. Each document can be in one of three different states: Draft, Moderation, and Published. With the State design pattern, we can create a `Document` class that can transition between these states and exhibit different behaviors in each state.



States and Behaviors:

Draft:

- In the Draft state, the document is not yet published.
- When the `publish` method is called, it transitions the document to the Moderation state.

Moderation:

- In the Moderation state, the document can be made public, but only if the current user is an administrator.
- When the `publish` method is called, if the user is an administrator, it transitions the document to the Published state.

Published:

- In the Published state, the document is immutable, and the `publish` method does nothing.

Implementation of the State Design Pattern: To represent these states, we create individual classes for each state: `Draft`, `Moderation`, and `Published`. Each class represents a state of the `Document` and implements behaviors specific to that state. The `Document` class holds a reference to the current state, and its behaviors are determined based on this reference.

This design allows us to add behaviors specific to document states and tailor the `publish` method for each state. It makes our system more flexible and extensible.

```
public class Document {
    private String state;

    public Document() {
        state = "draft";
    }

    public void publish() {
        switch (state) {
            case "draft":
                moveToModeration();
                break;
            case "moderation":
                approveForPublication();
                break;
            case "published":
                break;

        }
    }

    private void moveToModeration() {
        state = "moderation";
```

```
            System.out.println("Document moved to moderation status.");
        }

        private void approveForPublication() {
            state = "published";
            System.out.println("Document approved for publishable status.");
        }
    }
```

The biggest weakness of a state machine built with conditionals becomes apparent as we add more and more states and state-dependent behaviors to the Document class. The content of most methods will consist of complex conditionals that select the appropriate behavior of a method based on the current state. Such code has a structure that is challenging to maintain because any change in the transition logic may require altering state conditions in every method.

The problem tends to grow larger as a project evolves. It is quite challenging to predict all possible states and transitions during the design stage. Therefore, a lean state machine built with a limited set of conditionals can evolve into a bloated mess over time.

Let's take a look!

```
    interface DocumentState {
        void publish(Document document);
    }

    class DraftState implements DocumentState {
        @Override
        public void publish(Document document) {
            System.out.println("Document moved to moderation status.");
            document.setState(new ModerationState());
        }
    }
```

```
class ModerationState implements DocumentState {
    @Override
    public void publish(Document document) {
        System.out.println("Document approved for publishable status.");
        document.setState(new PublishedState());
    }
}

class PublishedState implements DocumentState {
    @Override
    public void publish(Document document) {
        System.out.println("The document has already been published.");
    }
}

public class Document {
    private DocumentState state;

    public Document() {
        this.state = new DraftState();
    }

    public void publish() {
        state.publish(this);
    }

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

This time, we created an interface called DocumentState to represent each state. Subsequently, we generated classes implementing each state (DraftState, ModerationState, PublishedState). We added a reference in the Document class to manage the state and designed the system by using the State Design Pattern when invoking the publish method to update the state. This allowed us to create classes for each state containing their specific behaviors, making the code more modular and easier to maintain.

Some of the advantages provided by the State Design Pattern are as follows:

1. **Modularity and Ease of Maintenance:** Classes representing each state encapsulate their specific behaviors. This allows us to handle each state separately and isolate state-specific code. Consequently, the code becomes more modular and easier to maintain.

2. **Flexibility and Extensibility:** Adding new states or modifying the behaviors of existing states is easier. To add a new state, you only need to create a new class and implement the interface. Similarly, modifying the behaviors of an existing state is done within the specific state class.

3. **Prevention of Code Duplication:** Since state transitions and state-dependent behaviors are encapsulated in separate classes, similar state transitions or behaviors can be reused. This results in less code duplication and cleaner code.

4. **Ease of State Transition:** Changing behaviors associated with a specific state or adding a transition to a new state is achievable by modifying or adding the class representing the relevant state. This facilitates ease of state transition.

5. **Clearer Code Understanding:** States and state-specific behaviors are clearly defined, making the code more understandable. Complex states and transitions are distinctly separated among classes.

The State Design Pattern, by offering these advantages, promotes a design that is modular, extensible, and easier to comprehend and maintain.

In this article, we explored the solutions that the State Design Pattern brings to the complexity of software development and state management.

This pattern suggests extracting state-specific codes into separate classes. By creating a class for each state and placing state-dependent behaviors within these classes, you can add new states or modify existing state behaviors by simply changing or adding the class representing the relevant state. This approach allows you to manage states independently and reduce maintenance costs.

With the State Pattern, state transitions can be controlled by both State classes and Context classes. In other words, a state change can be controlled not only by State classes but also by Context classes.

Wishing it contributes to your design and hoping it provides a new perspective for software development. Happy coding!

State Design Pattern Design Patterns