



# INVESTIGATING OBJECT-ORIENTED ANALYSIS AND DESIGN PATTERNS ALONG WITH ASSOCIATED SOLUTIONS

*Chukwudi Jeremiah Paul<sup>1</sup>, Onwe Festus Chijioko<sup>2</sup>*

<sup>1</sup> University of Bolton, UK

[cjp2crt@bolton.ac.uk](mailto:cjp2crt@bolton.ac.uk)

<sup>2</sup> University of Port Harcourt, Choba, Rivers State.

[festus.onwe@uniport.edu.ng](mailto:festus.onwe@uniport.edu.ng)

## ABSTRACT :

Design patterns in software engineering provide solutions to general design problems that improve both code reliability and versatility as well as maintenance capabilities. The investigation explores structural, behavioral, and creational design patterns to determine their function in producing straightforward routes for entity association implementation. Authors examine structural patterns because these designs illustrate the proper connections between objects and classes to design extensive system frameworks. Pattern development emerges from entity communication patterns through behavioral pattern analysis. The deployment of this communication improves through established patterns that simplify its implementation processes. The creational pattern group determines object creation methods that establish proper development frameworks for project object implementation. Design patterns go through evaluation, which combines goal assessment with analysis of code methods and technical specifications, as well as an assessment of how they address various design problems. This study examines how design patterns should be chosen by performing comparative research that aligns with particular software requirements. Software development leads to efficient programming outcomes because developers connect business requirements with technical solutions to create appropriate solutions from patterns.

**Keywords:** Object-Oriented Analysis, Design patterns

## INTRODUCTION

Design Pattern in software engineering is a known, trusted answer to a common problem in software design. This is not an already-made design that can be changed straight to code. It has to do with the interpretation or template of steps to resolve a problem that can be deployed in several different conditions. Design Patterns play an important role in software development as they make provision for solutions to be reused for common design issues. It supports scalability, maintainability, and efficiency of code by providing good approaches to solving the Problem. Software developers deploy these accepted and standard practices to implement more code that is manageable and scalable. It gives standard terminology and is peculiar to specific problems and scenarios.(Marinescu, 2002).

Patterns can't be copied into the Program code, just as it happens with off-the-shelf functions. It is not a particular piece of written code but rather a generally acceptable concept for resolving a specific problem in coding. You can follow the pattern information and develop a solution that is in line with your program. Algorithms are most of the time confused with patterns because the two techniques explain solutions to a few known problems. The algorithm provides a step-by-step way to follow and arrive at a specific goal and a pattern, as well as to implement a solution in a high-level description. Implementation of the same pattern is added to the double different programs, which may end up not being the same. An example of an algorithm is some ingredient for food preparation: The two have procedures to arrive at a goal. The objectives of this study are to give a clear understanding of design patterns and discuss their functions in software quality and development efficiency improvement. It will identify the types of design patterns, discuss each with its advantages and disadvantages, and highlight its limitations.(El Boussaidi and Mili, 2012)

## REVIEW ON DESIGN PATTERN

"The idea of design patterns originated from Christopher Alexander, an architect who introduced patterns in his book *"A Pattern Language"* (1977). He proposed that reusable architectural patterns could help design cities and buildings in a structured way".(Dawes and Ostwald, 2017) . Software development methodologies started to formalize during the 1970s while reusable solutions remained undeveloped that decade. The paradigm of object-oriented programming (OOP) was developing through the creation of programming languages that included Simula (1967) and Smalltalk (1972). The tested patterns of design, known as design patterns, function as proven solutions to software problems. Object-oriented methods are implemented in design patterns to create dependable, reusable solutions that solve design-related issues. The main objective of design patterns is to implement extensible approaches that control loose coupling. The use of design patterns brings about designs that reduce program components' interdependency while improving their extension and maintenance capabilities. The improper system design that adds to implementation time becomes an issue disturbing the

system performance according to antipattern dictation. The scenarios with problems that are documented with their proposed solutions act as correct documentation to aid developers to recognize and run away from errors and other design problems that impact performance.

Software designers use systematic solutions to handle frequent problems within software development which leads to better code clarity together with improved maintainability and operational efficiency. Software developers employ design patterns as code-writing templates to create programs that minimize time wastage and maximize clarity and system inspection capabilities. "In general, all software patterns can be classified as generative and non-generative. Both generative and non-generative patterns could be classified to be design patterns, organization patterns, analysis patterns, etc., depending on the aspect of software. Design patterns are the most known and used patterns today, as they are patterns in software engineering and reflect both low-level strategies for the design of components in the system and high-level strategies that impact the design of the overall system".(Tešanovic, 2005)

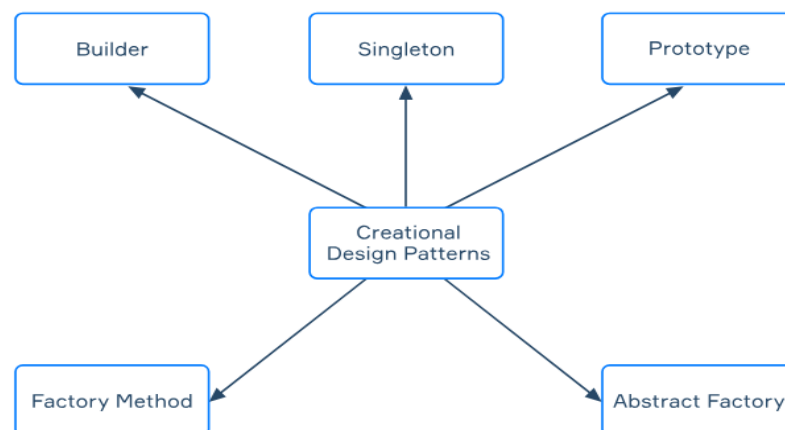
## SIGNIFICANCE OF DESIGN PATTERNS

1. Code reusability enables developers to put a stop to needless development through established solutions addressing common software development problems by using design patterns.
2. Using Design Patterns allows built software to grow through new functional abilities and increased efficient processing capabilities.
3. The architectural design enables developers to read and understand code by using standard terminology.
4. Existing solutions enable developers to prevent redundant work through their application of functional code to various problems, thus enabling code reuse.
5. Design Patterns supply adaptable components for loose connections through their deployment of adaptable system components.

## CREATIONAL DESIGN PATTERN

Creational design patterns in software engineering tend to deploy the object creation method, objects are being implemented in a condition suitable for the project. The real nature of an object implementation may lead to problems in the design, or complexity may be added to the design. This issue is solved by creational design patterns through handling the object creation. Any of the creational pattern types has its techniques, merits, and demerits. The following are the five forms of creational design patterns.(Dhait *et al.*, 2024)

### Types of creational patterns



<https://hyperskill.org/learn/step/16251>

Figure 1: Types of Creational Design Pattern

## FACTORY METHOD

Purpose – this pattern type implements the abstract class for an object creation that permits modification of object creation in subclasses. (Temaj, 2023)

### Suitable Scenarios

- ✓ If the object implementation logic has to be centralized
- ✓ When a particular kind of object to be implemented is decided at runtime

### Benefits

- ✓ It enhances the loose coupling between object creation and client code
- ✓ It supports scalability by approving new object types with no modification of already existing code

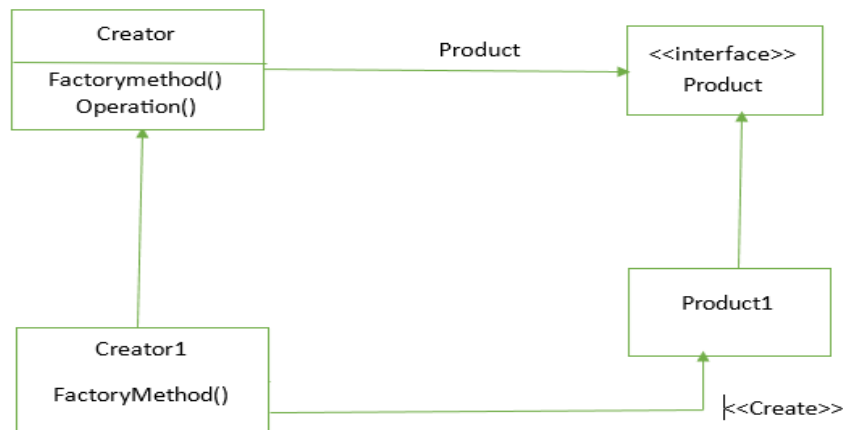


Figure2: Sample Class Diagram for Factory Method

**ABSTRACT FACTORY**

Purpose - Accepts using a method that creates other factories. This can be used further as a factory method.

Suitable Scenarios

- ✓ This is when concrete classes are not specified during multiple related object creation
- ✓ If multiple environments need to be supported by a system

Benefits

- ✓ Allows consistency between objects that are part of the same family
- ✓ Makes object implementation simple for complex applications

**PROTOTYPE**

Purpose - This has to do with copying a ready-made class to create a new one. The number of Subclasses that differ only in the technique they are initialized with their objects.

Suitable Scenarios

- ✓ If object implementation is costly and needs to be reduced
- ✓ If similar attributes with instances are always being created by an application

Benefits

- ✓ In object implementation, overhead is reduced
- ✓ Object structures are reserved while supporting modification

**BUILDER**

Purpose - allows the use of step-by-step methods to implement complex objects using simple objects.

Suitable Scenarios

- ✓ It is good when multiple optional components with complex objects are needed to be constructed
- ✓ When object creation needs to follow a step-by-step method

Benefits

- ✓ Code readability is enhanced by clearly explaining an object's construction
- ✓ Various configurations are used to handle object implementation

**SINGLETON**

Purpose - This is a creational pattern that implements one single instance of an object. in this process, a point of global access is created to this instance.

Suitable Scenarios

- ✓ When it enhances the connectivity of the database and configuration managers
- ✓ When it stops multiple instances of objects that influence system-wide operations

Benefits

- ✓ Memory is saved by stopping multiple unnecessary instances
- ✓ Grant global access point for steady object management

**Class Diagram for Creational Design Pattern**

In the creational design pattern, an object has to deploy instantiation to another object, while a class varies to the class that is instantiated by the use of inheritance. **For example**, let us overlook many details of the exact components of a house and focus on how it is built. The house is said to be a set of rooms. A room knows its neighbors, which may include another room, a door to another, or a wall. Only a few important parts of creating a house are identified. The diagram below identifies the connection between these classes.

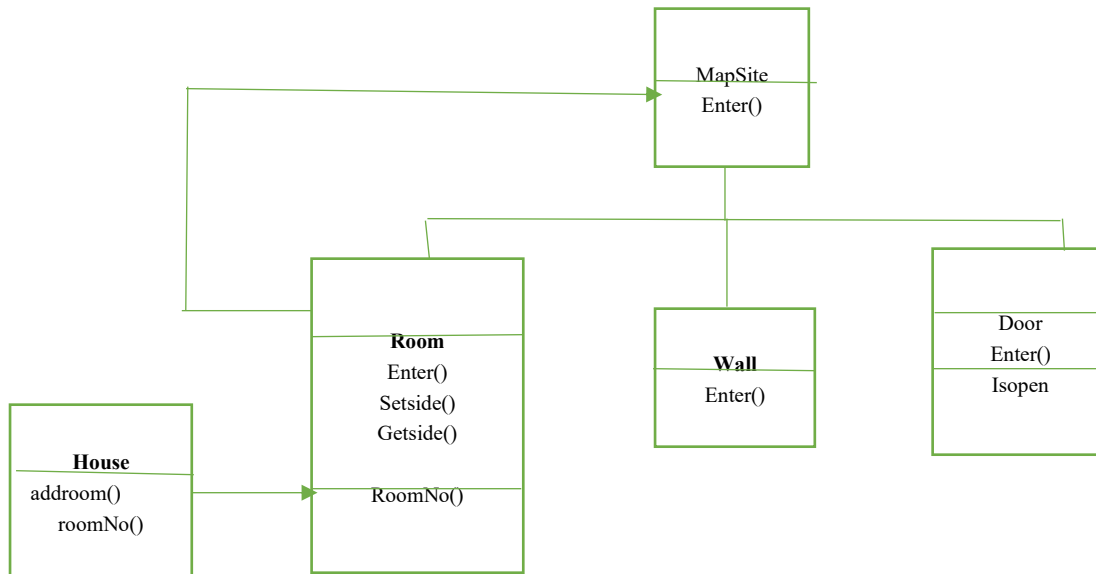


Figure 3: Class Diagram of creational pattern

### Code Implementation

```

Main.java  CreationalPatternDemo.java
interface Room {
    void describe();
}
// Concrete Room types
class LivingRoom implements Room {
    public void describe () {
        System.out.println("This is a Living Room with a cozy atmosphere.");
    }
}
// Concrete Room types
class Bedroom implements Room {
    public void describe() {
        System.out.println("This is a Bedroom with a comfortable bed.");
    }
}
// Factory to create rooms
class RoomFactory {
    public static Room createRoom(String type) {
        if (type.equalsIgnoreCase("LivingRoom")) {
            return new LivingRoom();
        } else if (type.equalsIgnoreCase("Bedroom")) {
            return new Bedroom();
        }
        return null;
    }
}
  
```

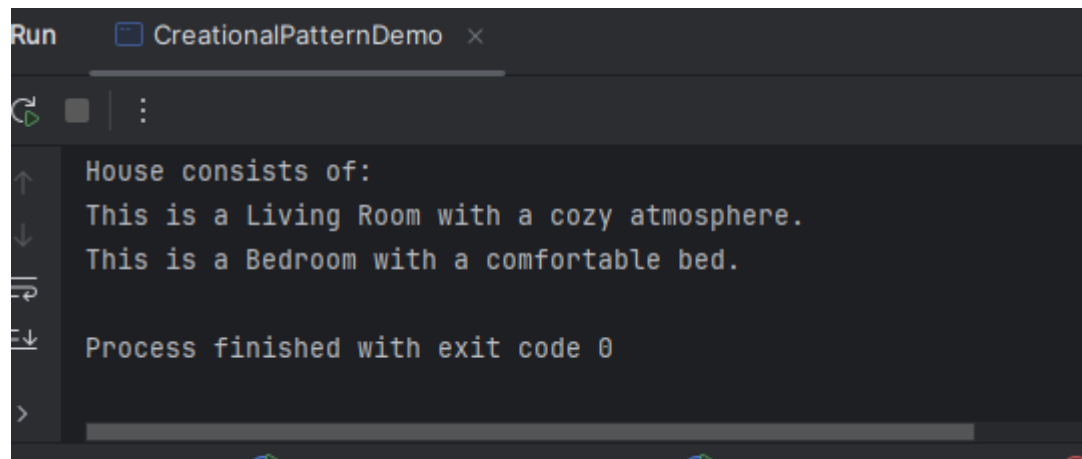
```

// House class
class House { 2 usages
    private Room room1; 2 usages
    private Room room2; 2 usages
    public House() { 1 usage
        // Using Factory to create rooms
        this.room1 = RoomFactory.createRoom( type: "LivingRoom");
        this.room2 = RoomFactory.createRoom( type: "Bedroom");
    }
    public void describeHouse() { 1 usage
        System.out.println("House consists of:");
        room1.describe();
        room2.describe();
    }
}

// Main Class
> public class CreationalPatternDemo{
>     public static void main(String[] args) {
        House house = new House();
        house.describeHouse();
    }
}

```

#### Output



```

Run  CreationalPatternDemo x
House consists of:
This is a Living Room with a cozy atmosphere.
This is a Bedroom with a comfortable bed.

Process finished with exit code 0

```

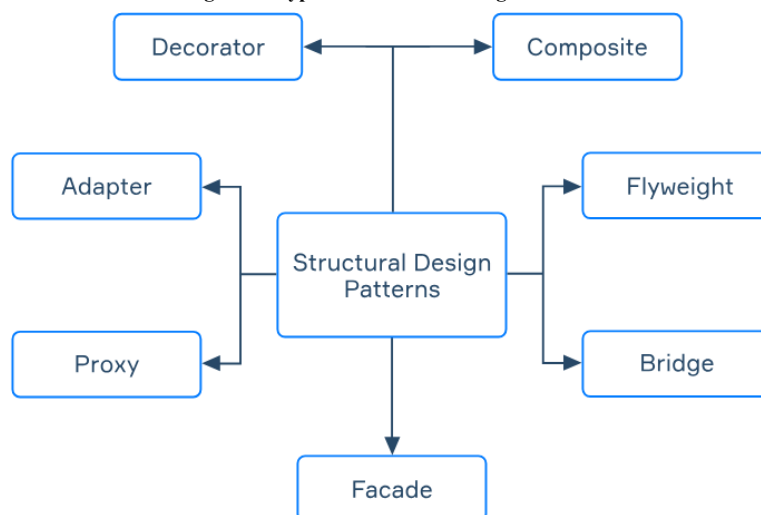
### STRUCTURAL DESIGN PATTERN

This type of design pattern eases the design by pointing out an easy route to implement the relationship between entities. The interest in structural patterns is to know how objects and classes are joined together to come out with large structures. Inheritance is been used in structural class patterns to join interfaces. A better idea of whether we need to compose, inherit, or create and maintain any other relationships between components. (Fontana, Maggioni and Raibulet, 2013)

#### ROLES OF STRUCTURAL DESIGN PATTERNS

- ✓ Complex structures are formed by joining objects in an efficient and flexible way
- ✓ In promoting loose coupling, dependencies are managed among classes
- ✓ Reusability is possible by allowing objects to be adapted without alteration of the code
- ✓ Maintainable object structures and scalability allow the system to be more adaptable

Figure 4: Types of Structural Design Pattern



<https://hyperskill.org/learn/step/17649>

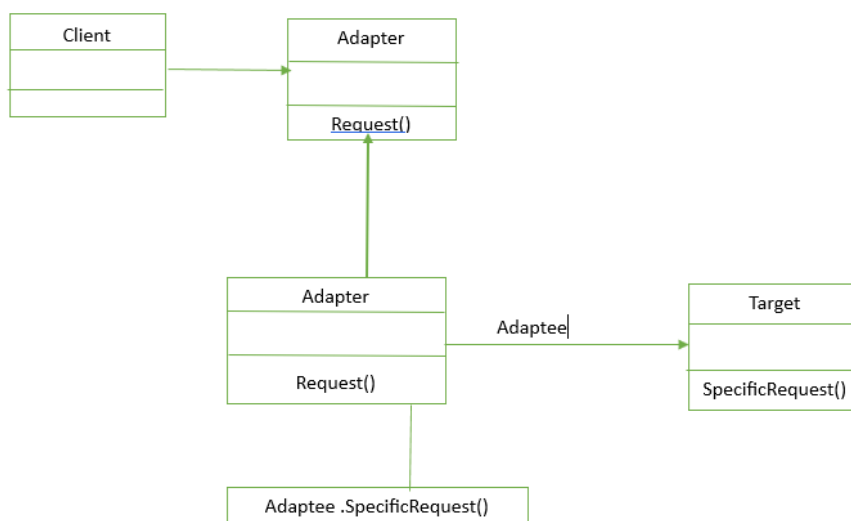
#### ADAPTER

Purpose - Looking at the name, you understand that it concerns joining objects with incompatible interfaces. (Harrer *et al.*, 2008)

#### How it Supports Collaboration

- ✓ One interface is permitted to work with another by bridging incompatibility system

Figure5: Sample class Diagram for Adapter



#### Bridge

Purpose - The Separation of abstraction and implementation of class, making it possible for their independent development, is supported through the bridge.

#### How it Supports Collaboration

- ✓ Abstraction and implementation can be allowed to look different without depending on each other

#### Composite

Purpose – This has structured objects in a hierarchical method that permits the client to manipulate each of them

#### How it Supports Collaboration

- ✓ It supports the same method to manage individual objects and compositions

#### Decorator

Purpose – This type of structural pattern improves object behavior with no original object alteration through the help of a special wrapper

#### How it Supports Collaboration

- ✓ Permits objects to acquire more features dynamically

#### Facade

Purpose – This makes available an interface for a complex set of objects (Ivanov and Sato, 2024)

#### How it Supports Collaboration

- ✓ An easy, unified interface is implemented for communication with complex systems

#### Flyweight

Purpose – the number of objects is added in this pattern to help fit into storage by sharing and by using their common parts

#### How it Supports Collaboration

- ✓ An efficient Object shearing is guaranteed

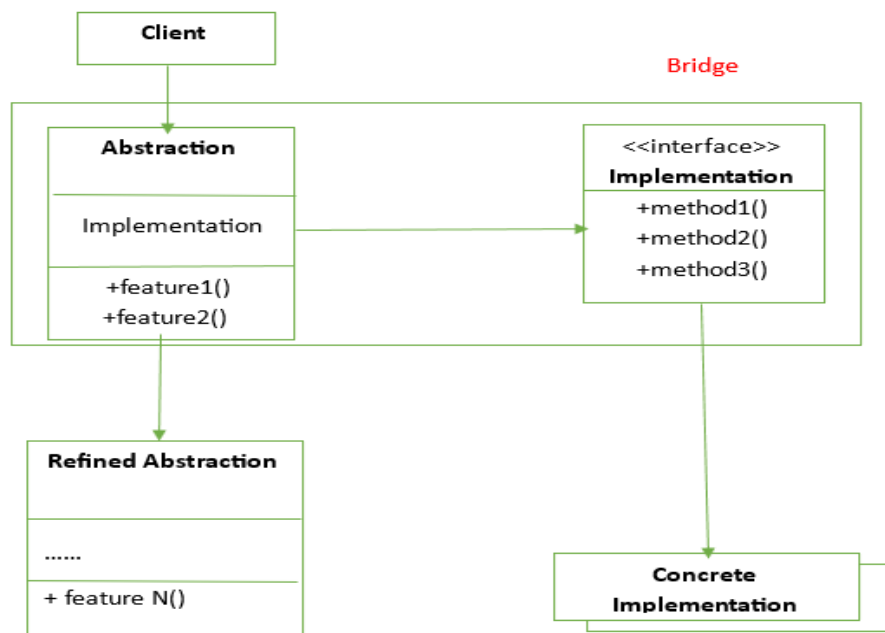
**Proxy**

Purpose – This makes a placeholder for another complex object available.

**How it Supports Collaboration**

- ✓ Underlying resources are given controlled and optimized access

**Figure 6: Class Diagram for Structural Design Pattern**



**Example:** Have a Coffee base class which we need to extend with Milk and Sugar features without touching its original structure.

**Code Implementation**

```

// Step 1: Create the Coffee Interface
interface Coffee { 7 usages 4 implementations
    String getDescription(); 6 usages 4 implementations
    double getCost(); 6 usages 4 implementations
}

// Step 2: Implement a Basic Coffee class
class BasicCoffee implements Coffee { 1 usage
    public String getDescription() { 6 usages
        return "Basic Coffee";
    }

    public double getCost() { 6 usages
        return 5.0;
    }
}

// Step 3: Create an abstract CoffeeDecorator class
abstract class CoffeeDecorator implements Coffee { 2 usages 2 inheritors
    protected Coffee coffee; 7 usages
    public CoffeeDecorator(Coffee coffee) { 2 usages
        this.coffee = coffee;
    }
}
  
```

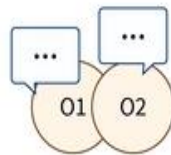




## BEHAVIORAL DESIGN PATTERN

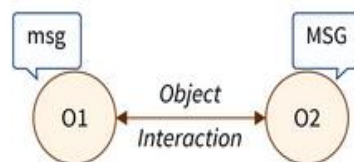
Behavioral design patterns. In this process, the patterns add flexibility in deploying this communication. This pattern follows the step that objects in an object-oriented project have to be connected in a way that complex implementation can be avoided and user input can be nicely arranged. Loose coupling techniques are deployed by the behavioral pattern to ensure flexibility and a good flow of information. In software engineering, when classes and objects are weakly connected, it is called loosely coupled object-oriented software systems. Because of this weak connection between classes and objects it is not as effective as those in tightly coupled systems. Being independent and reusable is possible for objects in loosely coupled systems because any change made has a small impact on the existence of another (Pettit IV and Gomma, 2006). The components of a behavioral design pattern are listed below

### Object interaction in tight coupling



*Inefficient Interaction without Behavioural Design Patterns*

### Object interaction in loose coupling



*Flexible Interaction using Behavioral Design Patterns*

<https://www.oodeesign.com/behavioral-patterns/>

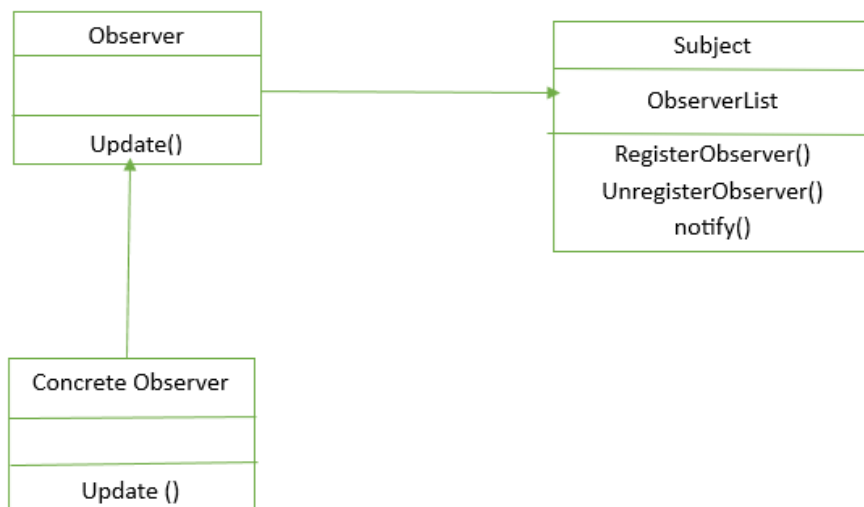
#### Observer Pattern

Purpose – provides one to many dependencies among objects, making sure that if an object changes its original state, all its connected dependents are signaled (Monday and Monday, 2003)

#### How it facilitates Communication

- ✓ Support automated feedback across multiple parts with no direct dependencies

Figure 7: Class Diagram for observer design pattern



## 5.2 Strategy Pattern

Purpose – The family of algorithms is defined, and different classes are encapsulated, permitting objects to change algorithms dynamically

### How it facilitates Communication

- ✓ permits objects to change character dynamically with no modification of the client code

### Command Pattern

Purpose – Objects are encapsulated as requests, using operations to be queued, parameterized, or logged for undo features

### How it facilitates Communication

- ✓ The sender is decoupled from the receiver by doing so. Flexible command execution is allowed

### Interpreter Pattern

Purpose – Grammar is defined, and an interpreter is made available to implement the sentences in a particular Language

### How it facilitates Communication

- ✓ A step-by-step technique to evaluate organized and structured expressions is provided

### Class Diagram of Behavioral Design Pattern

The new design features a notification system that enables news agencies to broadcast the latest updates to numerous news readers through the system. The system requires adaptive features for extensions while using a behavioral observer design pattern to manage relationships between news agencies and their newsreaders

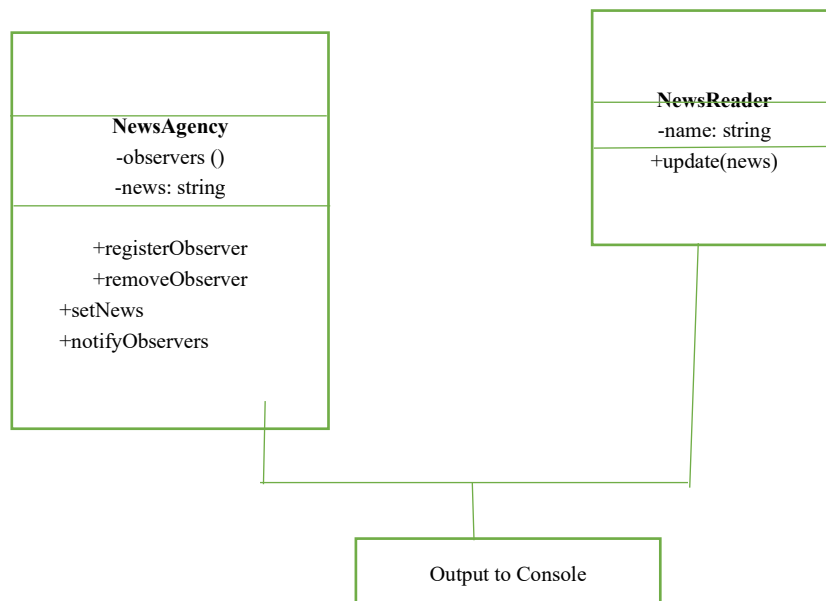


Figure 8: class Diagram of Behavioral Design Pattern

### Code Implementation

```

import java.util.ArrayList;
import java.util.List;
interface NewsReader { 10 usages 2 implementations
    void update(String news); 1 usage 2 implementations
}

// Step 2: Implement the subject (Observable) Interface
interface NewsAgency { 1 usage 1 implementation
    void subscribe(NewsReader reader); 2 usages 1 implementation
    void unsubscribe(NewsReader reader); 1 usage 1 implementation
    void notifyReaders(String news); 2 usages 1 implementation
}

// Step 3: Create a Concrete NewsAgency (Publisher)
class BBCNews implements NewsAgency { 2 usages
    private List<NewsReader> readers = new ArrayList<>(); 3 usages

    public void subscribe(NewsReader reader) { 2 usages
        readers.add(reader);
        System.out.println("A new reader has subscribed.");
    }

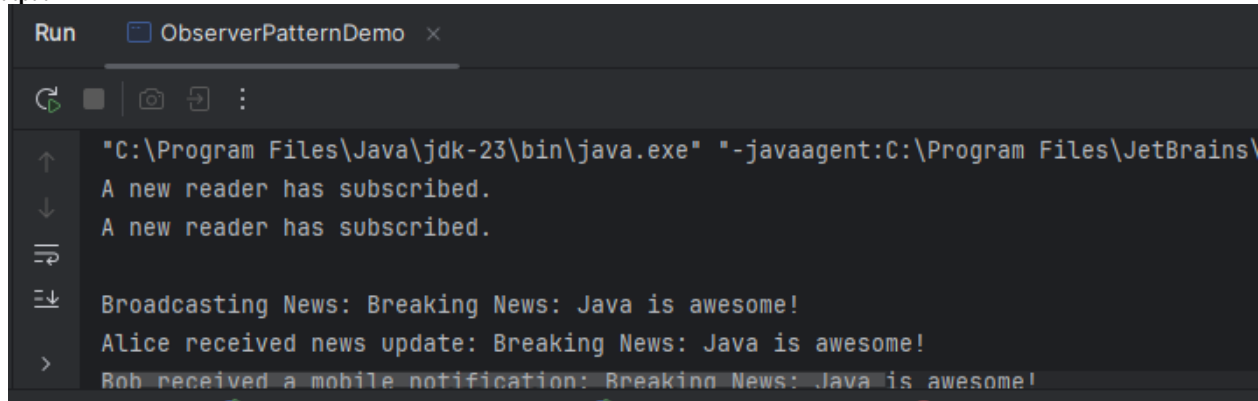
    public void unsubscribe(NewsReader reader) { 1 usage
        readers.remove(reader);
        System.out.println("A reader has unsubscribed.");
    }
}
  
```

```
public void notifyReaders(String news) { 2 usages
    System.out.println("\nBroadcasting News: " + news);
    for (NewsReader reader: readers) {
        reader.update(news);
    }
}

/ Step 4: Create Concrete Observers (Subscribers)
class OnlineSubscriber implements NewsReader { 1 usage
    private String name; 2 usages
    public OnlineSubscriber(String name) { 1 usage
        this.name = name;
    }
    public void update(String news) { 1 usage
        System.out.println(name + " received news update: " + news);
    }
}

class MobileSubscriber implements NewsReader { 1 usage
    private String name; 2 usages
    public MobileSubscriber(String name) { 1 usage
        this.name = name;
    }
    public void update(String news) { 1 usage
        System.out.println(name + " received a mobile notification: " + news);
    }
}

// Step 5: Test the Observer Pattern
public class ObserverPatternDemo {
    public static void main(String[] args) {
        // Create a news agency
        BBCNews bbc = new BBCNews();
        // Create subscribers
        NewsReader alice = new OnlineSubscriber(name: "Alice");
        NewsReader bob = new MobileSubscriber(name: "Bob");
        // Subscribe to BBC News
        bbc.subscribe(alice);
        bbc.subscribe(bob);
        // Publish news
        bbc.notifyReaders(news: "Breaking News: Java is awesome!");
        // Unsubscribe a reader and publish another news
        bbc.unsubscribe(bob);
        bbc.notifyReaders(news: "Latest Update: Observer Pattern in action!");
    }
}
```

**Output**


```

Run  ObserverPatternDemo x
"C:\Program Files\Java\jdk-23\bin\java.exe" -javaagent:C:\Program Files\JetBrains\
A new reader has subscribed.
A new reader has subscribed.
Broadcasting News: Breaking News: Java is awesome!
Alice received news update: Breaking News: Java is awesome!
Bob received a mobile notification: Breaking News: Java is awesome!

```

**COMPARATIVE ANALYSIS OF CREATIONAL, STRUCTURAL, AND BEHAVIORAL DESIGN PATTERNS****Characteristics and Purpose of Creational Design Pattern**

The design pattern focuses on object implementation to achieve flexible and reusable code from existing implementations. The pattern enables the system to be dependent on implementation but not on organizational presentation or object structure. (Jenila and Ranjana, 2011)

**Strength**

- ✓ The hidden complex instantiation logic provides encapsulation to developers.
- ✓ The environment modification enables developers to use objects (Flexibility)
- ✓ The decrease in dependency occurs when implementing concrete systems (Decoupling).

**Weakness**

- ✓ During setup and maintenance, there is always more complexity (**Overhead**)
- ✓ Code complexity may be multiplied by the factory and abstract factory (**May lead to excess Abstraction**)

**Characteristics and Purpose of Structural Design Pattern**

It focuses on organizing classes and objects to come out with huge structures while making sure of flexibility and efficiency in relationships.

**Strengths**

- ✓ Structures are made reusable and separate concerns (**Enhance code maintainability**)
- ✓ Memory and control access are controlled by flyweight and proxy (**Improves performance**)
- ✓ A system architecture can be managed (**Scalability**)

**Weakness**

- ✓ Unwanted indirection is caused by multiple layers (Increased **Complexity**)
- ✓ Execution can be slow due to additional processing caused by proxy and adapter

**Characteristics and Purpose of Behavioral Design Pattern**

The goal of this pattern group is to improve object communication using flexible methods that increase encapsulation in message transfers.

**Strength**

- ✓ Flexibility and modularity appear while determining objects and their mutual interactions
- ✓ The processing of decoupling logic supports reusable, maintainable code through its functionality.
- ✓ The application shows advancements in command patterns along with mediator and chain of responsibility patterns.

**Weakness**

- ✓ High usage of UML increases the number of both object layers and interactions
- ✓ The implementation of strategies combined with state along with observer could lead to runtime overhead during dynamic operations

**Table 1: Comparative analysis of Design Pattern**

COMPARATIVE EVALUATION OF DESIGN PATTERNS			
Pattern Category	Best suited for	Strength	Weakness
Creational	It is good for the management of lifecycle and object instantiation	In concrete implementations, dependency is decreased	During setup and maintenance, there is always more complexity
Structural Pattern	Controls connections among objects	Memory and control access are controlled by flyweight and proxy	Unwanted indirection is caused by multiple layers
Behavioral	The management of objects during workflow functions remains of utmost importance	In the process of decoupling logic, reusable, maintainable code is supported	Runtime overhead may be introduced by state, observer, and strategy due to dynamic behavior

## CONCLUSION

The tested solutions that design patterns offer basic design problems constitute a fundamental aspect of software Engineering practice. Research focuses on three different design pattern categories known as behavioral and creational as well and structural. This study lists design pattern realization examples while identifying both their beneficial aspects and limiting features and particular traits. The implementation of entities becomes simpler through structural design patterns by showing standard methods to link different entities.

Different patterns in this assessment show their unique set of advantages and drawbacks. Developers gain the capacity to identify superior design patterns through understanding trade-offs which leads to superior software development results for their projects. Expertise in design patterns must be learned by Software Engineers to create software systems that merge both scalability attributes with maintainability features. Modern software development techniques need to apply Agile combined with DevOps practices to deliver maximum benefits in contemporary development.

## REFERENCES

1. Dawes, M.J. and Ostwald, M.J. (2017) Christopher Alexander's A Pattern Language: analysing, mapping and classifying the critical response. *City, Territory and Architecture*, [Online] 4 pp. 1–14 Available from: <https://> . [Accessed].
2. Dhait, S., Sapate, A., Gadge, A., Borkar, P., Badhiye, S. and Aher, U.B. (2024) Analysis Of The Best Creational Design Patterns In Software Development. In: . , [Online] Available from: <https://> . [Accessed].
3. El Boussaidi, G. and Mili, H. (2012) Understanding design patterns—what is the problem? *Software: Practice and Experience*, [Online] 42 (12), pp. 1495–1529 Available from: <https://> . [Accessed].
4. Fontana, F.A., Maggioni, S. and Raibulet, C. (2013) Design patterns: a survey on their micro-structures. *Journal of Software: Evolution and Process*, [Online] 25 (1), pp. 27–52 Available from: <https://> . [Accessed].
5. Harrer, A., Pinkwart, N., McLaren, B.M. and Scheuer, O. (2008) The Scalable Adapter design pattern: Enabling interoperability between educational software tools. *IEEE Transactions on Learning Technologies*, [Online] 1 (2), pp. 131–143 Available from: <https://> . [Accessed].
6. Ivanov, M. and Sato, J. (2024) Façade Design Pattern Optimization Workflow Through Visual Spatial Frequency Analysis and Structural Safety Assessment. *Journal of Facade Design and Engineering*, [Online] 12 (1), pp. 43–62 Available from: <https://> . [Accessed].
7. Jenila, P.A. and Ranjana, P. (2011) Design pattern prediction techniques: A comparative analysis. In: . , [Online] Available from: <https://> . [Accessed].
8. Marinescu, F. (2002) *EJB design patterns*. ed. [Online] : Wiley New York. Available from: <https://> . [Accessed].
9. Monday, P.B. and Monday, P.B. (2003) Implementing the Observer Pattern. *Web Services Patterns: Java™ Platform Edition*, [Online] , pp. 187–204 Available from: <https://> . [Accessed].
10. Pettit IV, R.G. and Gomaa, H. (2006) Modeling behavioral design patterns of concurrent objects. In: . , [Online] Available from: <https://> . [Accessed].
11. Temaj, G. (2023) Factory design pattern. Source: [https://www.researchgate.net/publication/350611051\\_Factory\\_Design\\_Pattern](https://www.researchgate.net/publication/350611051_Factory_Design_Pattern). Retrieved from, [Online] 28 pp. Available from: <https://> . [Accessed].
12. Tešanovic, A. (2005) What is a pattern. *Dr.ing.course DT8100 (prev.78901/45942/DIF8901) Object-oriented Systems*, [Online], pp. Available from: <https://> . [Accessed].