THE KNOWLEDGE UNIVERSITY

Lecture 8: Design Patterns – Part I

IT2030 – Software Engineering

Lesson Learning Outcomes

- Explain the concept and importance of design patterns in software development.

- Identify the key components of a design pattern.

- Describe the structure and intent of the Singleton and Observer patterns.

- Illustrate these two patterns with suitable real-world examples.

What is a Design Pattern?

- A proven solution to a common software design problem in a specific context.

- Provides a reusable template for solving recurring design issues.

- Helps developers communicate ideas clearly using a shared vocabulary.

- Inspired by Christopher Alexander (architect, 1977), and formalized in software by the Gang of Four (1994).

What is a Design Pattern?

- A problem that someone has already solved.
- A model or design to use as a guide.
- More formally: "A proven solution to a common problem in a specified context.’’

Real-World Examples:

- Blueprint for a house
- Manufacturing

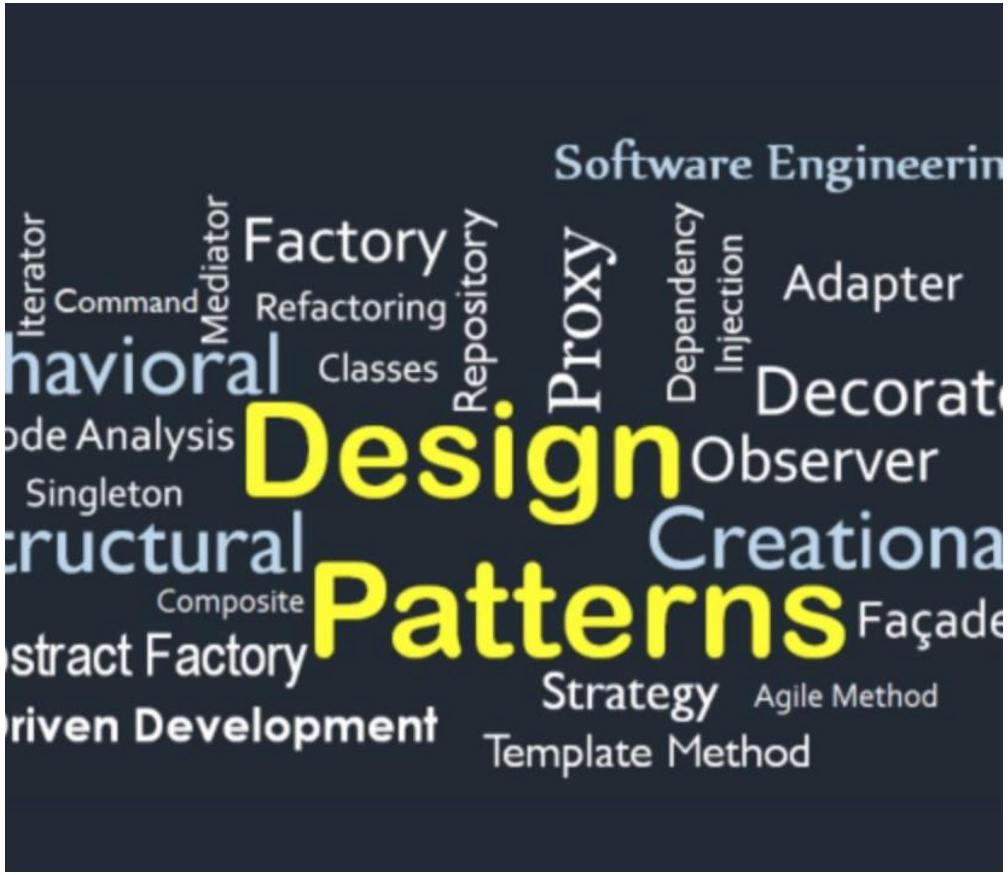
What is a Design Pattern?

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

-Christopher Alexander-

Why Study Design Patterns?

- Provides software developers with a toolkit for handling problems that have already been solved.
- Helps you *think* about how to solve a software problem.



Why Do We Need Design Patterns?

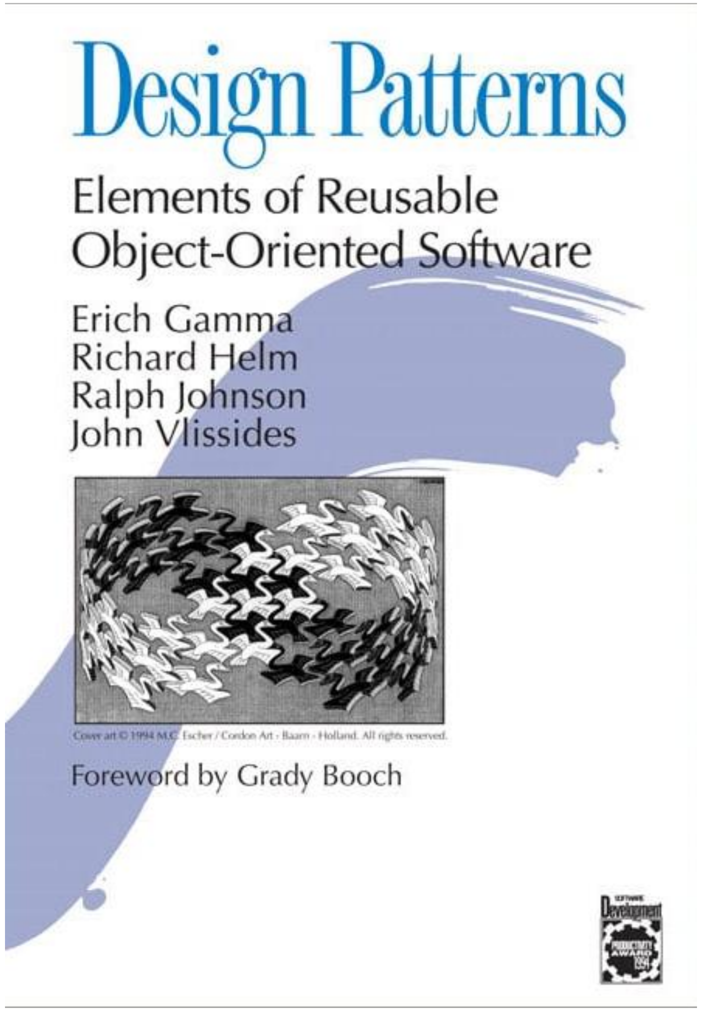
- **Proven Solution -** Design patterns give reliable solutions to common problems, so developers don’t have to reinvent the wheel.
- **Reusable** - Design patterns are adaptable and not limited to a single problem.
- **Expressive -** Design patterns are an elegant solution.
- **Prevent the Need for Refactoring Code -** Since the design pattern is already the optimal solution for the problem, this can avoid refactoring.
- **Lower the Size of the Codebase -** Each pattern enables system changes without full redesign and often uses less code.

Design Patterns

- The idea of **patterns** came earlier from **Christopher Alexander** (an architect, 1977) who used patterns in building architecture.
- In software, people had been informally using recurring solutions before 1994.
- The **Gang of Four (GoF)** in 1994 made the concept **formal and widely known** by cataloguing 23 patterns in their book.

The Gang of Four (GoF)

- Design patterns were popularized in software engineering by four authors in 1994, through their book
- Book: *Design Patterns: Elements of Reusable Object-Oriented Software*
- Authors:
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides
- Collectively known as the **Gang of Four (GoF)**



Key Components of a Design Pattern

1. Name
- Describes the pattern
- Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)

2. Problem
- Describes when to apply the pattern
- Answers - What is the pattern trying to solve?

3. Solution
- Describes elements, relationships, responsibilities, and collaborations that make up the design

4. Consequences
- Results of applying the pattern
- Benefits and Costs
- Subjective, depending on concrete scenarios

Design Patterns Classification

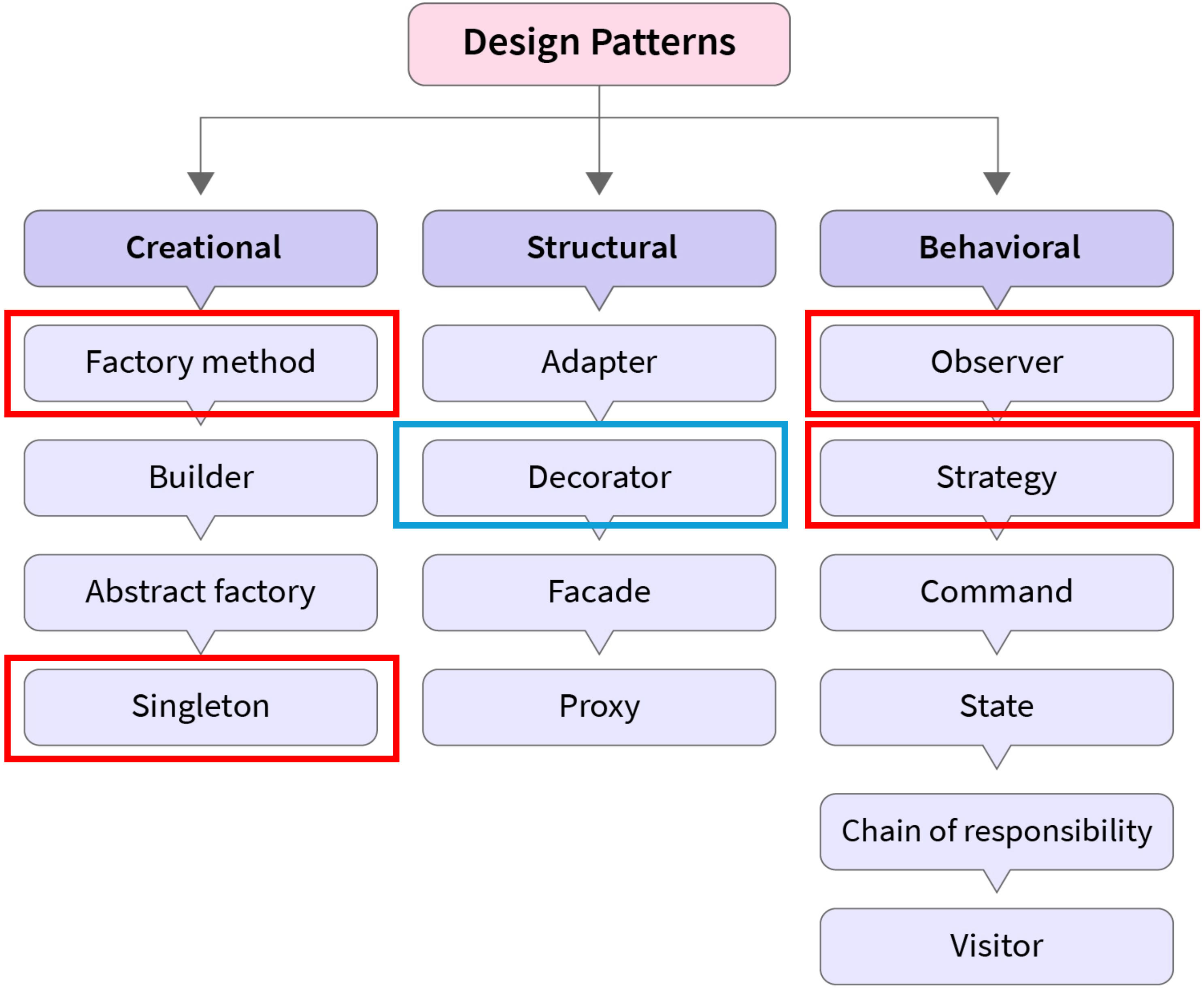
Design Pattern

- **CREARIONAL** — Deals with object creation and initialization, providing guidance about which objects are created for a given situation.
- **STRUCTURAL** — Deals with class and object composition, or how to assemble objects and classes into larger structures.
- **BEHAVIORAL** — Deals with communication between objects and how responsibilities are assigned between objects.

Design Patterns Classification

A Pattern can be classified as,

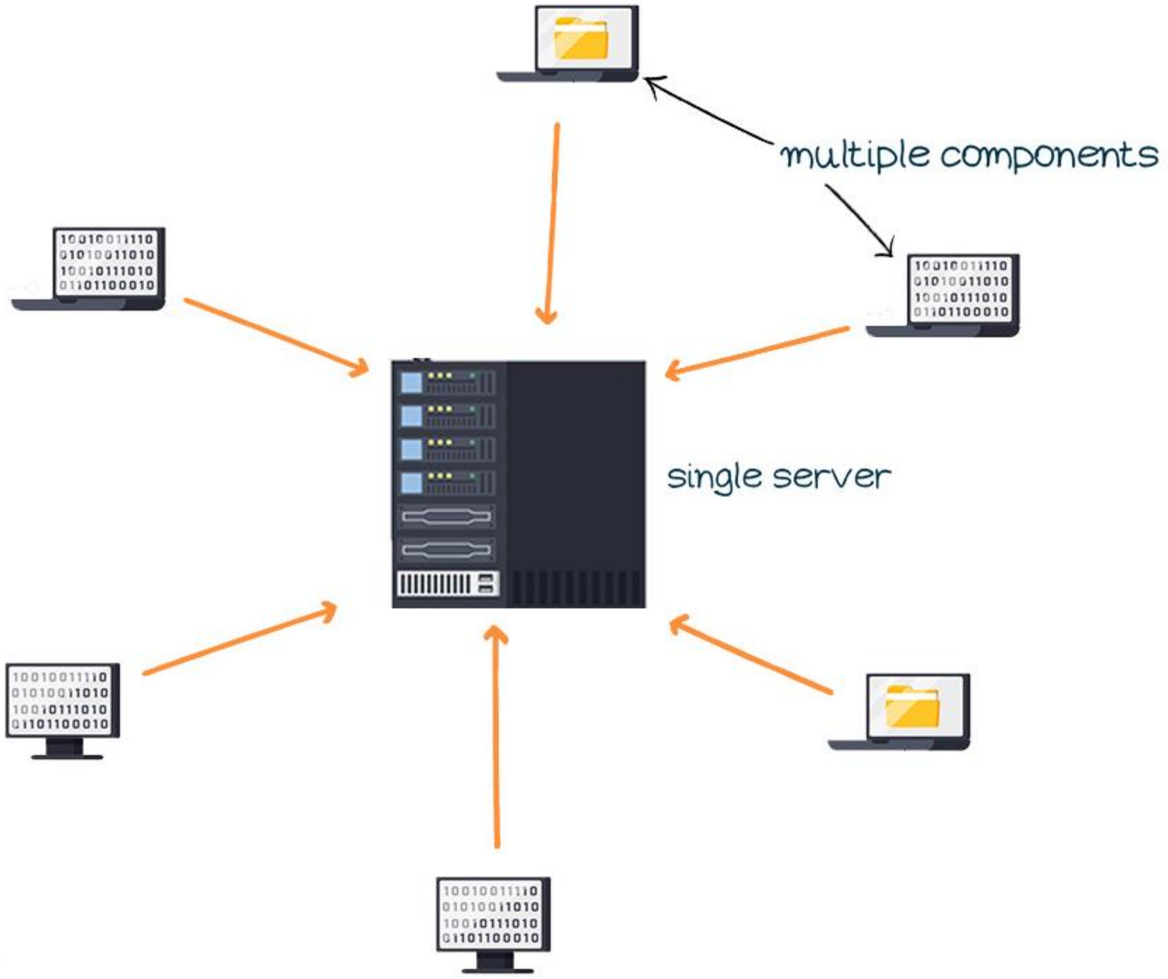
- **Creational –** Object creation
- **Structural –** Relationship between entities
- **Behavioral –** Communication between objects


Design Patterns
Creational
Structural
Behavioral
Factory method
Builder
Abstract factory
Singleton
Adapter
Decorator
Facade
Proxy
Observer
Strategy
Command
State
Chain of responsibility
Visitor

Creational: Singleton Pattern

Singleton Design Pattern

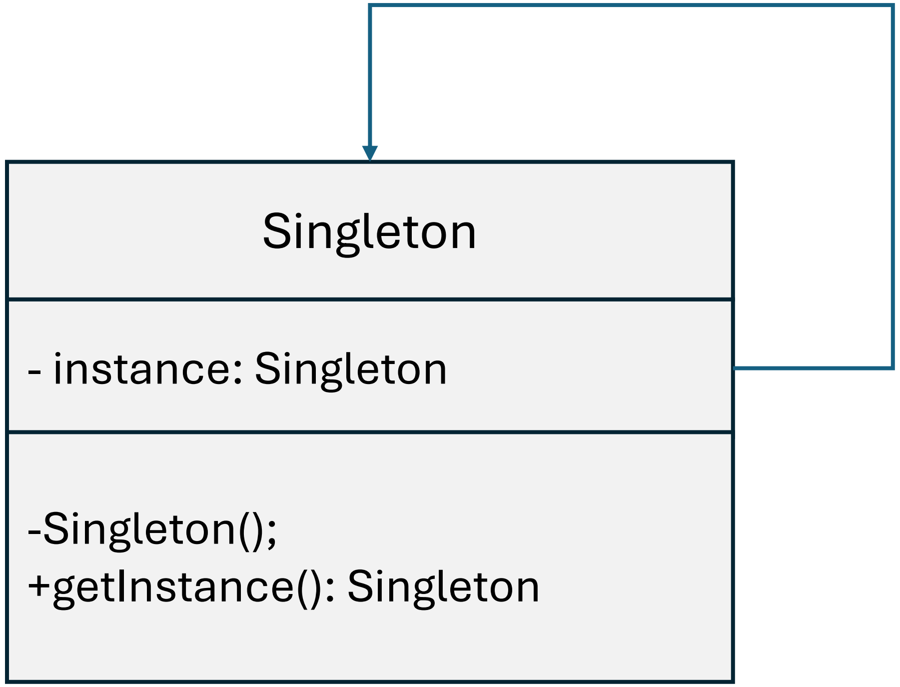
The singleton design pattern ensures that ***a class has only one instance during runtime and provides a global access point,*** allowing other classes to use its functionality without the need to instantiate it multiple times.



Singleton Pattern

- **Intent**: Ensure a class has only one instance and provide a global point of access to it.
- **Problem**: How can we guarantee that one and only one instance of a class can be created?
- **Solution**: Define a class with a **private constructor**. The class constructs a single instance of itself. Supply a **static method** that returns a reference to the single instance.

Singleton: Basic Structure



The Singleton class declares the static method ***getInstance*** that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the ***getInstance*** method should be the only way of getting the Singleton object.

How to create a Singleton class in Java?

Step#1: **Make the Constructor Private**

- Start by declaring the constructor as private.
- This prevents other classes from creating new objects of the Singleton class.
- If the class has more than one constructor, mark each one as private.

```java
public class Singleton {

  //private constructor to prevent instantiation
  private Singleton() {}
}
```

How to create a Singleton class in Java?

Step#2: **Declare a Static Instance Variable**

- A private static instance variable of the same class that is the only instance of the class.

```
public class SingletonClass {

  //static instance variable
  private static SingletonClass instance;

}
```

How to create a Singleton class in Java?

Step#3: **Provide a Static Method to Get the Instance**

- Declare a static method with the return type as an object of this singleton class.

```
public class SingletonClass {

  public static SingletonClass getInstance() {
    if (instance == null) {
     instance = new SingletonClass ();
    }
    return instance;
  }
}
```

Complete Example of a Singleton Class

```
public class SingletonClass {

  private static SingletonClass instance;          // Static instance variable

  private SingletonClass () {}               // Private constructor to prevent instantiation

  public static SingletonClass getInstance() {
    if (instance == null) {
     instance = new SingletonClass ();
    }
    return instance;
  }
}
```

Example Usage

<table>
<tr><th>How to use it</th><th>How it works</th></tr>
<tr><td>

```
public class SingletonTest {
  public static void main(String[] args) {

    // Get the singleton instance
    Singleton obj1 = Singleton.getInstance();
    Singleton obj2 = Singleton.getInstance();

    // Check if both instances are the same
    System.out.println(obj1 == obj2); // true
  }
}
```

</td><td>

First call:

- When you call Singleton.getInstance() for the first time,
the object is created → **new Singleton().**

Second call onwards:

- It **does NOT create a new object/instance**, it **returns the same old object/ instance**.

Result:

- No matter how many times you call it, you always get the **same instance**.

</td></tr>
</table>

Real-World Examples

- The President of a country

A country can only have one president at a time, and whenever a decision needs to be made by the president alone. The president in this example represents a singleton.

Real-World Examples – The Government

A country can have only one official government. Regardless of the personal identities of the individuals who form governments, the title, “The Government of Sri Lanka”, is a global point of access that identifies the group of people in charge.

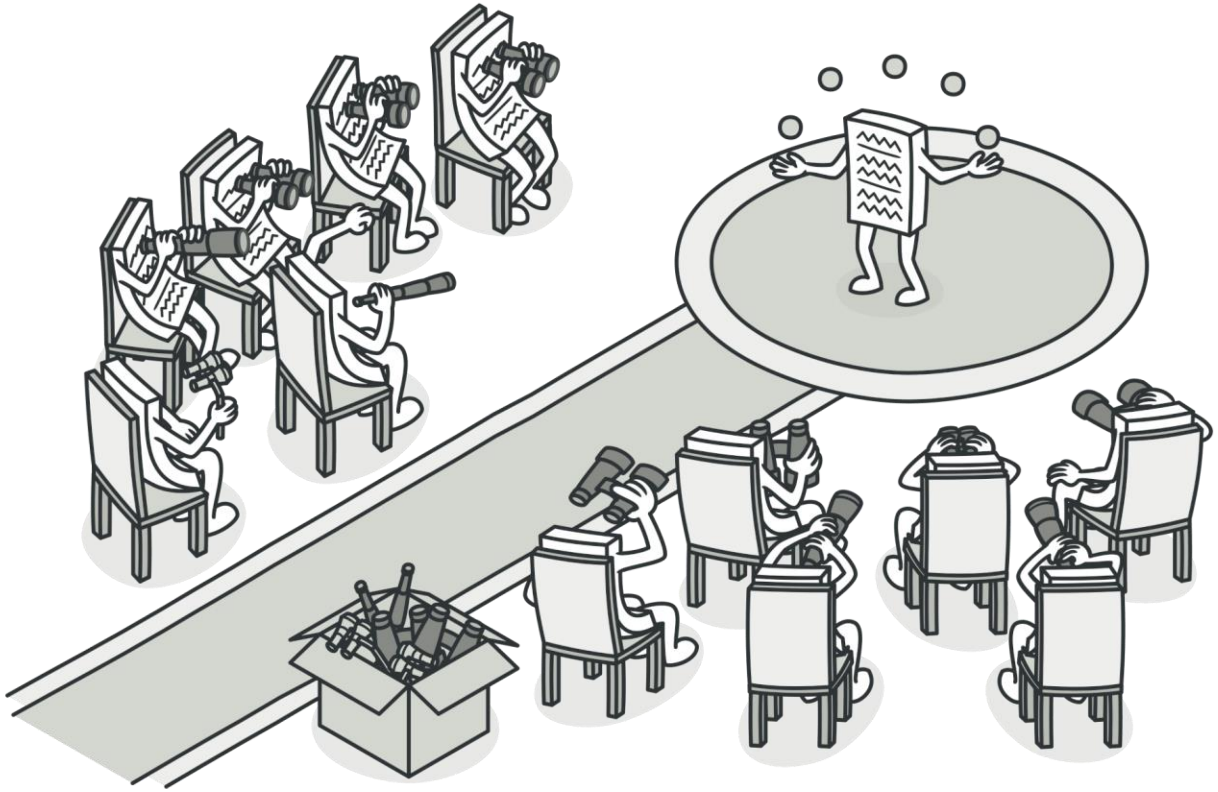
- **Advantages of Singleton Class**
  - **Resource Effectiveness**: Saves resources by reusing the same instance.
  - **Global Access of the Instance**: Provides a global point of access to the instance.
  - **Thread Safety**: Can be implemented to ensure safe concurrent access.

- **Disadvantages of Singleton Class**
  - **Difficulty in Testing**: Singleton classes can be difficult to test due to their global state.
  - **Possibility of Tight Coupling**: Can lead to tight coupling between classes.
  - **Potential for Excessive use**: Excessive use of singletons can lead to poor design and maintainability issues.

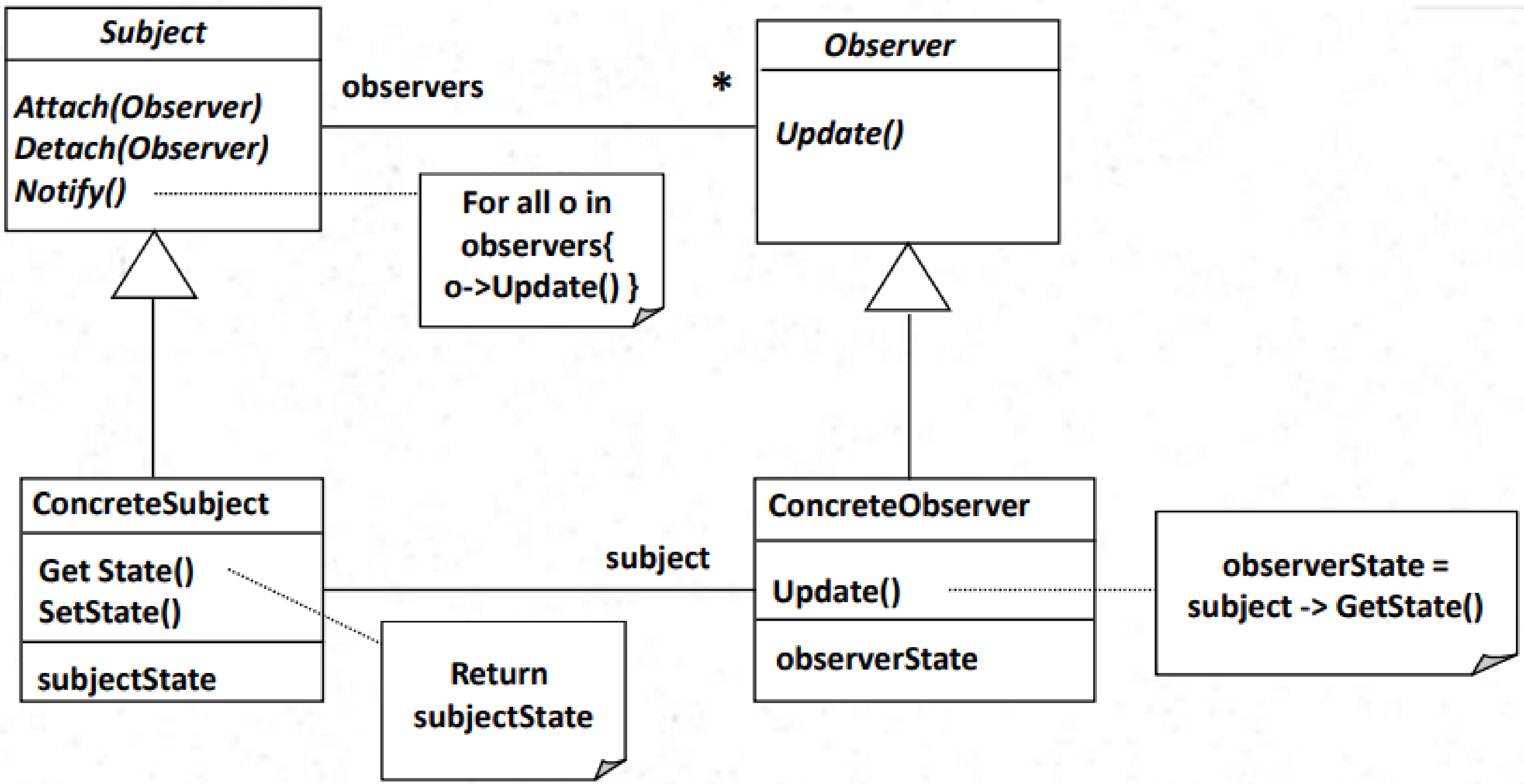
Behavioral: Observer Pattern

Definition

Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

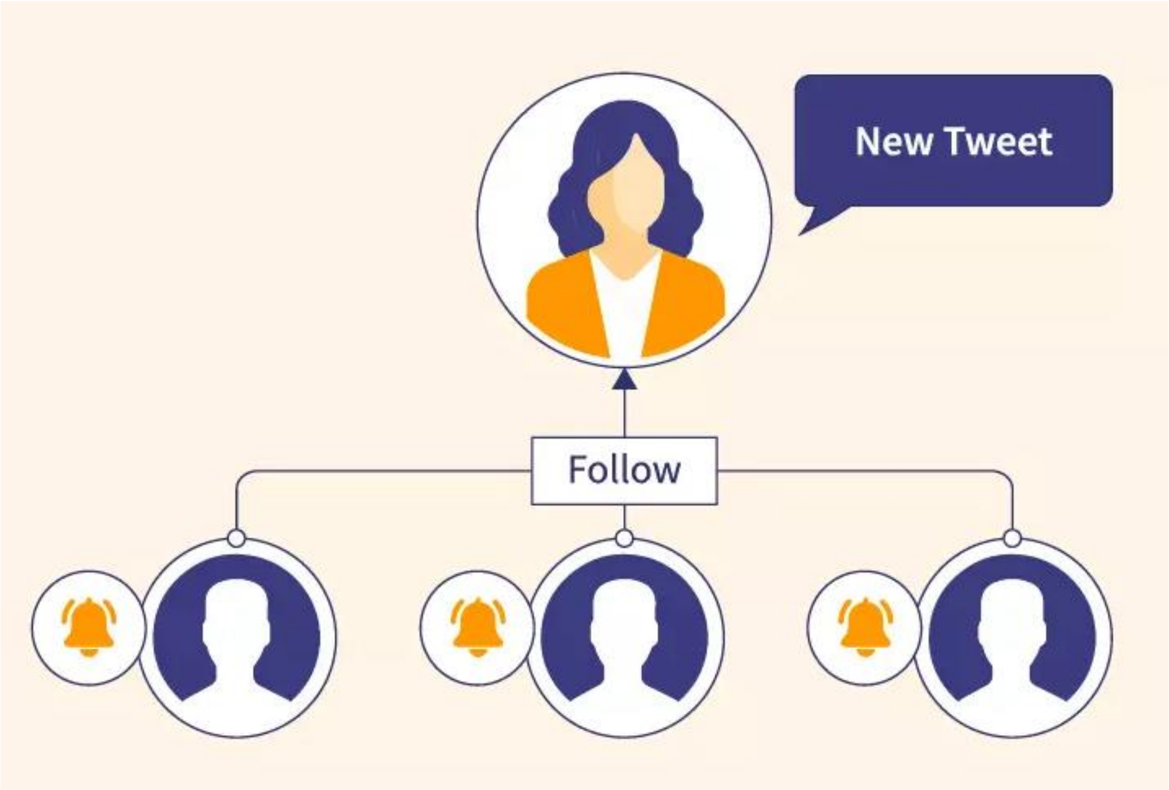


The Structure of Observer



Real-world Examples of Observer pattern

Any **social media platform**, such as Facebook or Twitter, can be a real-world example of an observer pattern. When a person updates their status, all their followers get a notification. A follower can follow or unfollow another person at any point in time. Once unfollowed, a person will not get notifications from the subject in the future.

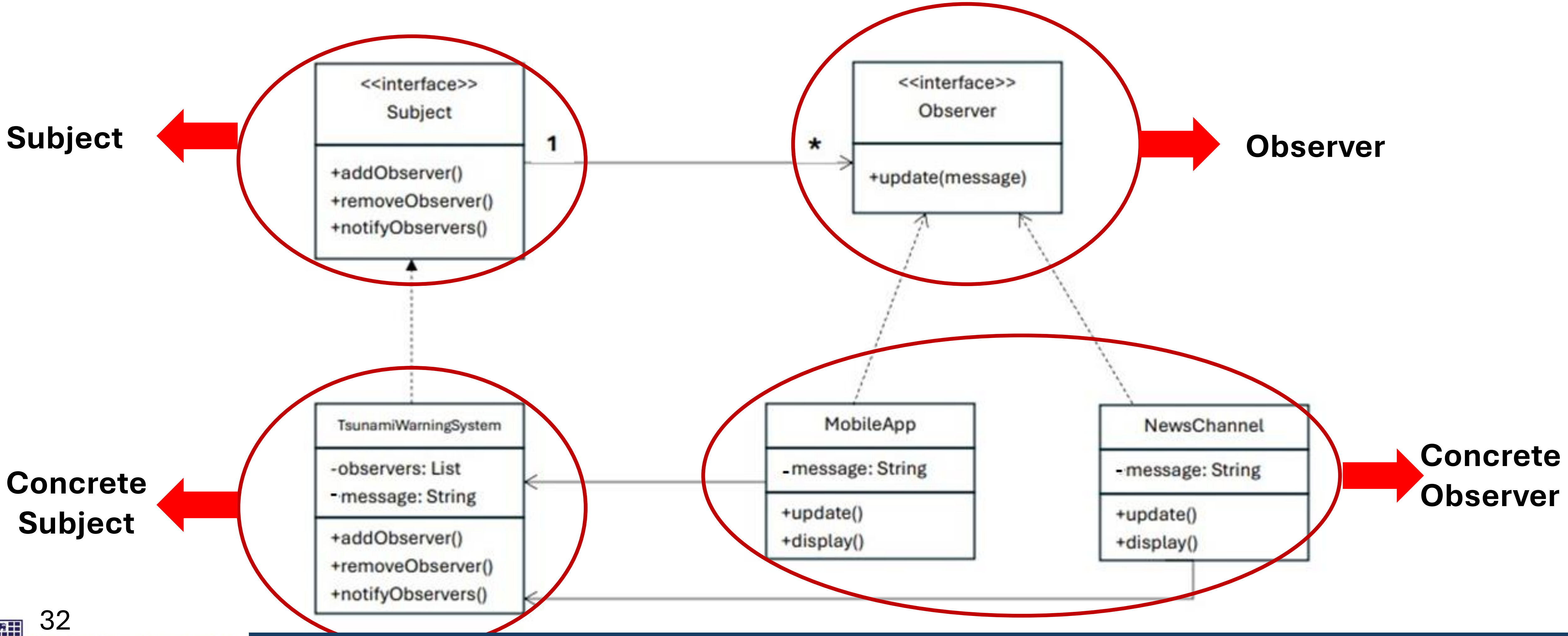


Components of Observer Design Pattern

- **Subject**: Maintains a list of observers, provides methods to add/remove them, and notifies them of state changes.
- **Observer**: Defines an interface with an update() method to ensure all observers receive updates consistently.
- **ConcreteSubject**: A specific subject that holds actual data. On state change, it notifies registered observers.
- **ConcreteObserver**: Implements the observer interface and reacts to subject updates

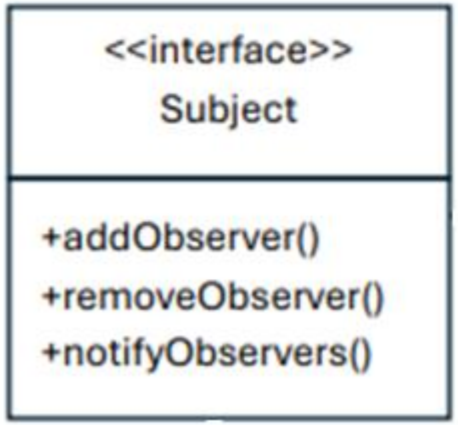
How does the Observer Design Pattern use?

Example: Tsunami Warning System – Class Diagram



Implementation - Subject Class

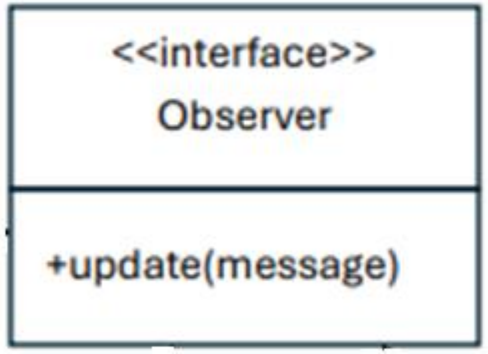
- The "**Subject"** interface outlines the operations a subject should support.
- **"addObserver"** and "**removeObserver"** are for managing the list of observers.
- **"notifyObservers"** is for informing observers about changes.




```
public interface Subject {
   void addObserver(Observer observer);
   void removeObserver(Observer observer);
   void notifyObservers();
}
```

Implementation - Observer Class

- The **"Observer"** interface defines a contract for objects that want to be notified about changes in the subject (“**TsunamiWarningSystem"** in this case).
- It includes a method **"update"** that concrete observers must implement to receive and handle updates.

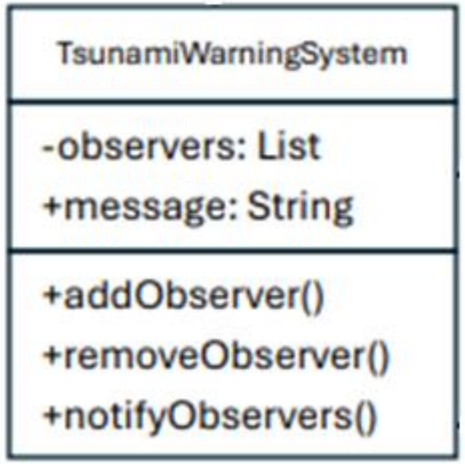



```
public interface Observer {
   void update(String message);
}
```

Implementation - ConcreteSubject (TsunamiWarningSystem)

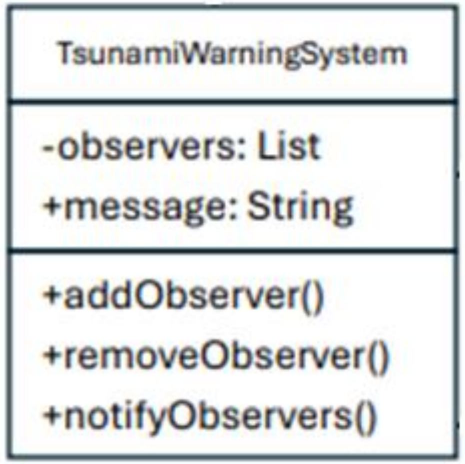
- **“TsunamiWarningSystem"** is the concrete subject implementing the "**Subject"** interface.
- It maintains a list of observers ("**observers"**) and provides methods to manage this list.
- **"notifyObservers"** iterates through the observers and calls their "**update"** method, passing the current weather.
- **"setMessage"** method updates the condition and notifies observers of the change.

Implementation – ConcreteSubject (TsunamiWarningSystem)




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

// Observer interface
interface Observer {
   void update(String message);
}

// Subject interface
interface Subject {
   void addObserver(Observer observer);
   void removeObserver(Observer observer);
   void notifyObservers();
}
```

Implementation – ConcreteSubject (TsunamiWarningSystem)




```java
// Concrete Subject
class TsunamiWarningSystem implements Subject {
   private List<Observer> observers = new ArrayList<>();
   private String message = "";

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

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

   @Override
   public void notifyObservers() {
      for (Observer observer : observers) {
         observer.update(message);
      }
```

Implementation – ConcreteSubject (TsunamiWarningSystem)

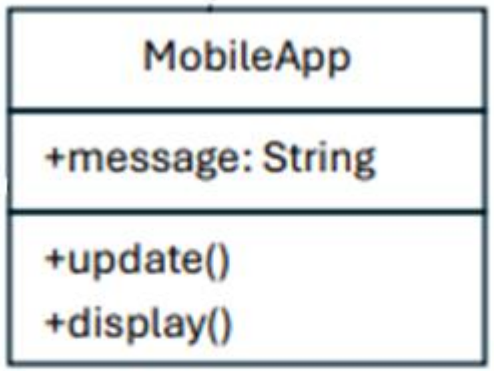
```
public void setMessage(String newMessage) {
    this.message = newMessage;
    notifyObservers();
}
```

We can use the setMessage() style, or getMessage() / setMessage()style as follows ; both are correct as long as the subject updates its value and notifies the observers.

```
public String getMessage() {
   return message;
}

public void setMessage(String newMessage) {
   this.message = newMessage;
   notifyObservers();
}
```

Implementation – ConcreteObserver (MobileApp)

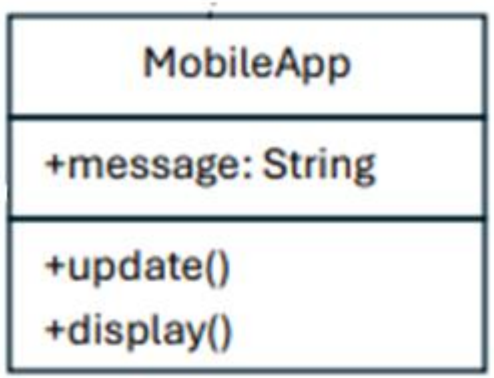
- **“MobileApp"** is a concrete observer implementing the "**Observer"** interface.
- It has a private field **message** to store the latest message.
- The "**update"** method sets the new message and calls the "**display"** method.
- **"display"** prints the updated message to the console.




```java
import java.util.Observer;
import java.util.Observable;

public class MobileApp implements Observer {
   private String message;

   @Override
   public void update(Observable o, Object arg) {
      if (arg instanceof String) {
         this.message = (String) arg;
         display();
      }
   }

   private void display() {
      System.out.println(“Mobile App: Message updated - " + message);
   }
}
```

Implementation – ConcreteObserver (MobileApp)

- **“NewsChannel"** is another concrete observer similar to **“MobileApp"**.
- It also implements the "**Observer"** interface, with a similar structure to “MobileApp”.




```
public class NewsChannel implements Observer {
   private String message;

   @Override
   public void update(String message) {
             this.message = (String) arg;
             display();
   }

   private void display() {
      System.out.println(“News Channel: Message
updated - " + message);
   }
}
```

Pros and Cons

Pros

- This design pattern allows information or data transfer to multiple objects without any change in the observer or subject classes.
- It adheres to the loose coupling concept among objects that communicate with each other.

Cons

- The Observer pattern can increase complexity and potentially cause efficiency issues if it's not executed properly.
- The fundamental shortcoming of this design pattern is that the subscribers/observers are updated in a random sequence.

Thank You