Design Pattern

BCS1430

Dr. Ashish Sai

📆 **Week 5 Lecture 2**

💻 **BCS1430.ashish.nl**

📍 **EPD150 MSM Conference Hall**

Behavioral Design Patterns

Behavioral patterns in software design focus on effective communication and the assignment of responsibilities among objects.

Behavioral Patterns

Pattern	Description	Covered
Observer	Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.	✅
Strategy	Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.	✅
Command	Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.	❌
State	Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.	❌
Chain of Responsibility	Passes the request along the chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.	❌
Interpreter	Provides a way to evaluate language grammar or expressions. The Interpreter pattern defines a grammar for the language, as well as an interpreter that uses the grammar to interpret sentences in the language.	❌
Memento	Captures and externalizes an object’s internal state so the object can be restored to this state later.	❌
Visitor	Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.	❌
Template Method	Defines the skeleton of an algorithm in the method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing its structure.	❌

Strategy Pattern

Strategy Pattern

Composition over inheritance!

Strategy Pattern is defined as:

- **Defining a family of algorithms**
- **Encapsulating each algorithm**
- **Making them interchangeable**

Inheritance vs. Composition

- Inheritance is not always intended for code reuse.
- Composition offers greater flexibility in many scenarios.
- Strategy Pattern focuses on using composition over inheritance.

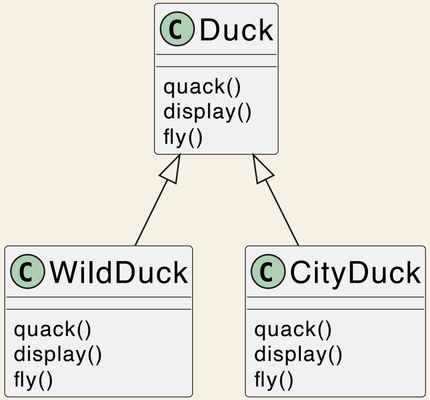
Problem Statement: Duck Example

- Consider a system with different types of ducks.
- Each duck type has its own display method.
- Common methods like `quack` are shared.

```java
public class Duck {
    public void quack() {
        // Common quack behavior
    }
    public abstract void display();
}
```

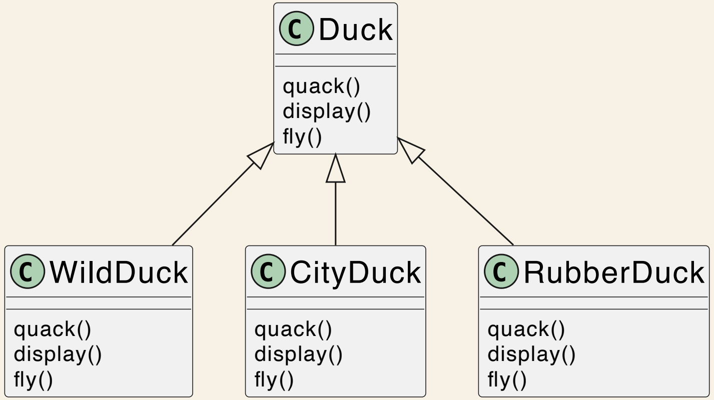
Problem Statement: Duck Example

- We have different types of ducks: wild duck, city duck, rubber duck, lets add them one by one



Problem Statement: Duck Example

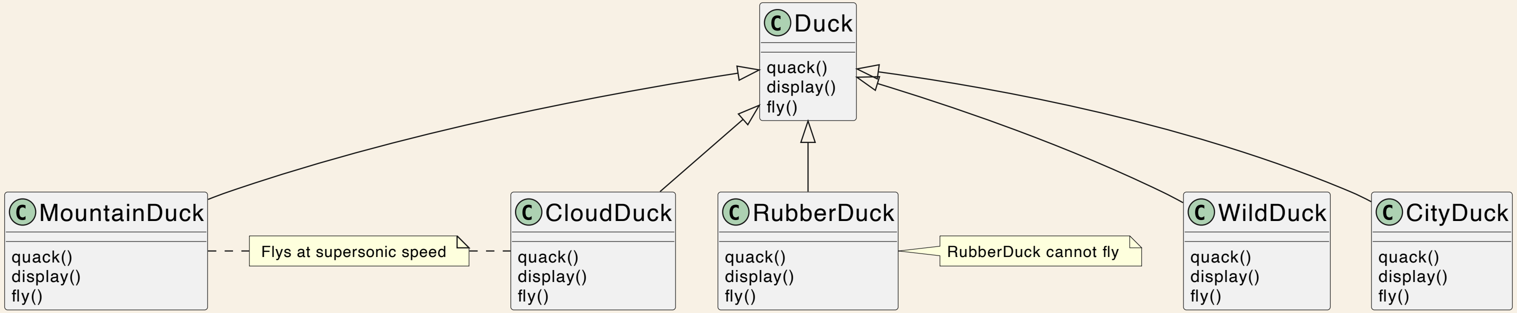
- `I want a rubber duck` 🐥



> `Can RubberDucks fly?`

Problem Statement: Duck Example

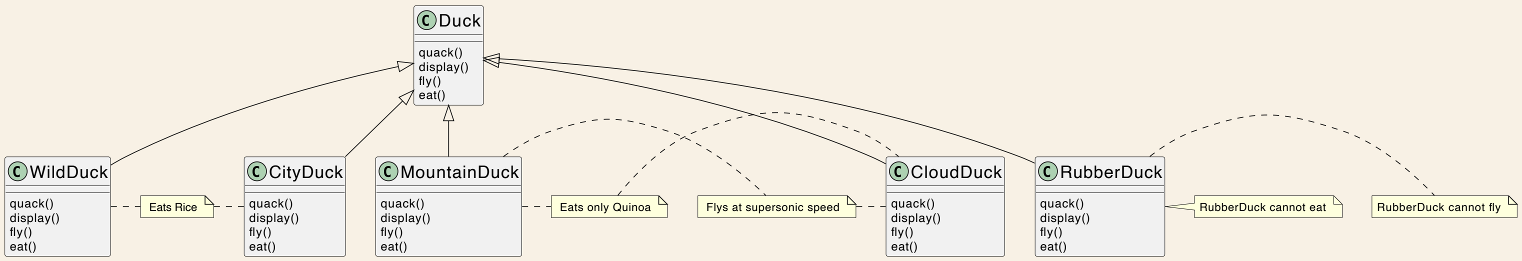
- We found two new types of ducks: MountainDuck and CloudDuck, they fly at supersonic speed [1] in a ZigZag pattern.



1. Really they don’t, don’t google it.

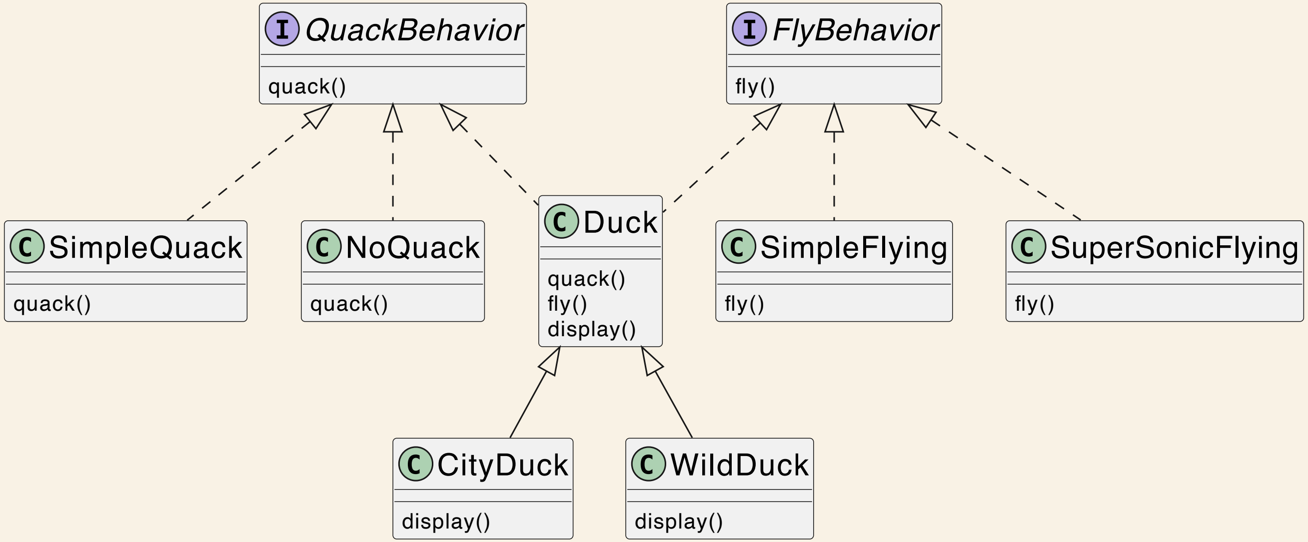
Problem Statement: Duck Example

- Now the Ducks want food, WildDuck and CityDuck eat rice but MountainDuck and CloudDuck eat only quinoa whereas RubberDuck eats ?



Introducing the Strategy Pattern

- The Strategy Pattern allows the duck's behaviors to vary independently.
- Encapsulates quacking and flying behaviors.



Strategy Design Pattern: Intent of Strategy Pattern

The intent of the Strategy pattern is to define a set of interchangeable algorithms or strategies that can be selected at runtime according to the needs of the context or client.

Strategy Design Pattern: Problem and Solution

- **Problem:** Need for a flexible way to incorporate different behaviors or algorithms within a class and the ability to change them at runtime.
- **Solution:** The Strategy pattern suggests separating the behavior into different strategy classes and using a reference to these strategies in the context class.

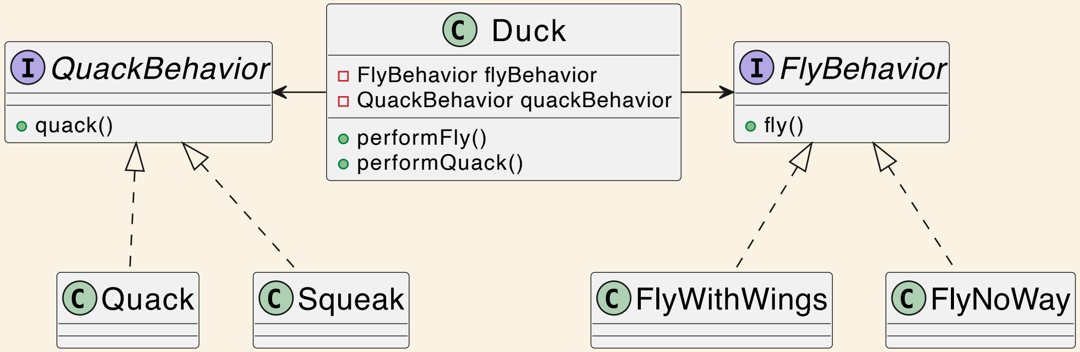
Problem with Inheritance: Adding Fly Method

- Adding `fly` method to Duck class leads to issues.
- Not all ducks should fly (e.g., rubber ducks).

```java
public class Duck {
    public void fly() {
        // Flying behavior
    }
}
```

Strategy Pattern Solution: Encapsulating Behaviors

- Separate `fly` and `quack` behaviors into different strategies.
- Each duck type can have its own flying and quacking behavior.



Implementing Duck Subclasses

- Different types of ducks inherit from Duck class.
- Each subclass implements its own display method.

```java
public class MountainDuck extends Duck {
    public MountainDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }
    public void display() {
        // MallardDuck specific display
    }
}
```

Advantages of Strategy Pattern

- Promotes flexible code structure.
- Allows behaviors to change dynamically.
- Reduces dependency on inheritance.

Decoupling Behaviors

- Behaviors are not hard-coded in the Duck class.
- They can vary independently from the duck type.

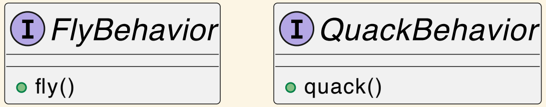
```java
public class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }
}
```

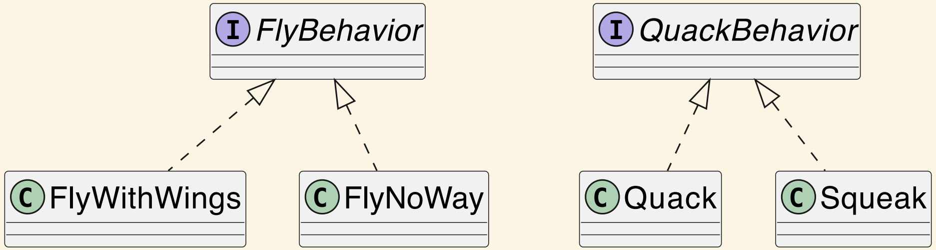
Defining Behavior Interfaces

- `Define interfaces for each behavior.`



Concrete Implementations

- Implement different flying and quacking behaviors.



Strategy Pattern in Duck Subclasses

- Subclasses of Duck can choose different behaviors.

```java
public class RubberDuck extends Duck {
    public RubberDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Squeak();
    }
    public void display() {
        // RubberDuck specific display
    }
}
```

Strategy Pattern: Flexibility

- Easy to add new behaviors without modifying existing classes.

```java
public class JetFlyingBehavior implements FlyBehavior {
    public void fly() {
        // Jet-powered flying
    }
}
```

Problem: Code Duplication in Inheritance

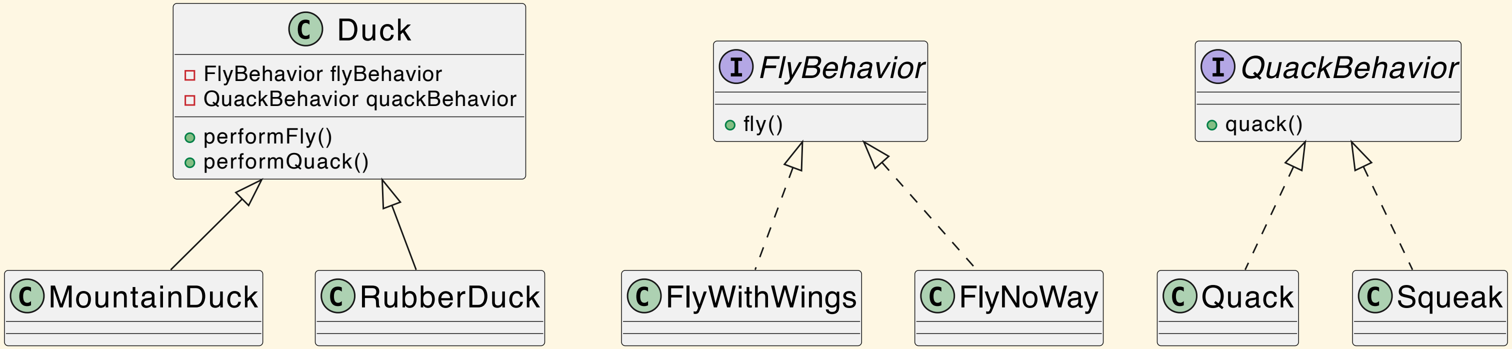
- `Inheritance can lead to duplicated code across subclasses.`

Solving Code Duplication

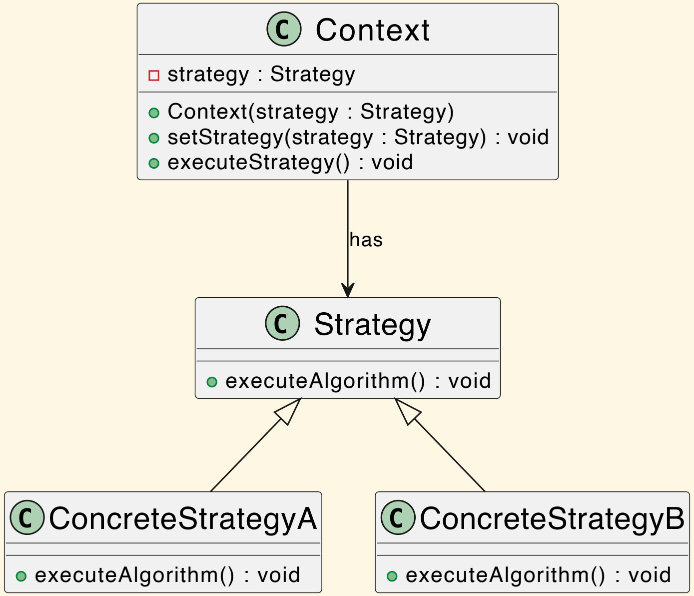
- `Strategy Pattern avoids duplication by sharing behavior implementations.`

Strategy Pattern in Context

- Allows ducks to have various combinations of behaviors.
- Easy to maintain and extend.



Strategy Design Pattern: Structure of Strategy Pattern



- **Context:** Maintains a reference to a Strategy object and delegates it the algorithm execution.
- **Strategy:** Common interface for all strategies defining the algorithm execution method.
- **ConcreteStrategy:** Implements the algorithm using the Strategy interface.

Strategy Design Pattern: Implementation in Java: Context and Strategy

```java
// Strategy Interface
interface Strategy {
    void executeAlgorithm();
}

// Context Class
class Context {
    private Strategy strategy;

    Context(Strategy strategy) {
        this.strategy = strategy;
    }

    void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    void executeStrategy() {
        strategy.executeAlgorithm();
    }
```

Strategy Design Pattern: Concrete Strategies in Java

```java
// Concrete Strategy A
class ConcreteStrategyA implements Strategy {
    public void executeAlgorithm() {
        // Implement algorithm A
    }
}

// Concrete Strategy B
class ConcreteStrategyB implements Strategy {
    public void executeAlgorithm() {
        // Implement algorithm B
    }
}
```

Strategy Design Pattern: Applicability

Use the Strategy pattern when:

- You have different variations of an algorithm and want to switch between them at runtime.
- You want to avoid exposing complex, algorithm-specific data structures.
- You want to replace inheritance with composition for behavioral variations.

Strategy Design Pattern: Pros and Cons

Pros:

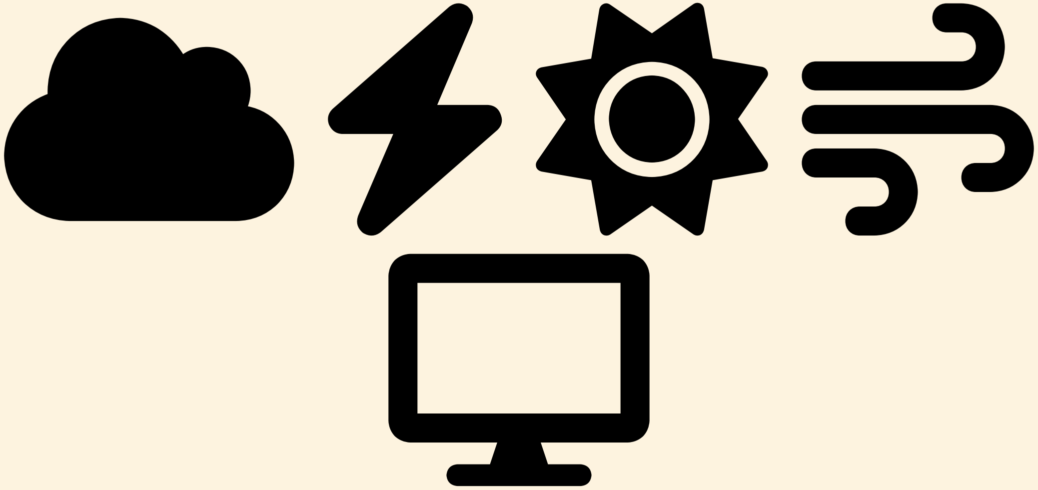
- Enables the Open/Closed Principle by allowing the introduction of new strategies without changing the context.
- Simplifies unit testing by isolating algorithms.

Cons:

- Increases the number of objects in the application.
- Clients must be aware of the differences between strategies to select the right one.

Observer Pattern

Weather Station and Display



Understanding the Problem

- **Scenario:** When an object changes its state, other objects need to be notified.
- **Challenge:** Continuously checking (polling) the state of an object is inefficient.

Basics of Observer Pattern

- **Definition:** A design pattern where an object, known as the subject, notifies a list of observers about its state changes.
- **Key Concept:** Push vs. Pull notification (move from pull to push).

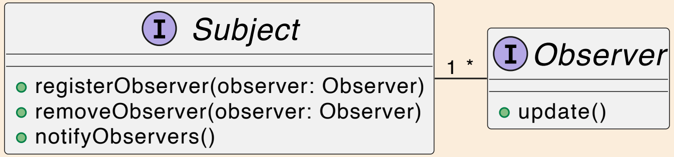
Observer Pattern: Intent

Observer pattern allows for the establishment of a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Observer Pattern: Problem and Solution

- **Problem:** Managing knowledge about changes in a system’s state can be complex when multiple entities need updates.
- **Solution:** Observer pattern offers a subscription model where subjects notify observers about changes, promoting decoupling and efficient data distribution.

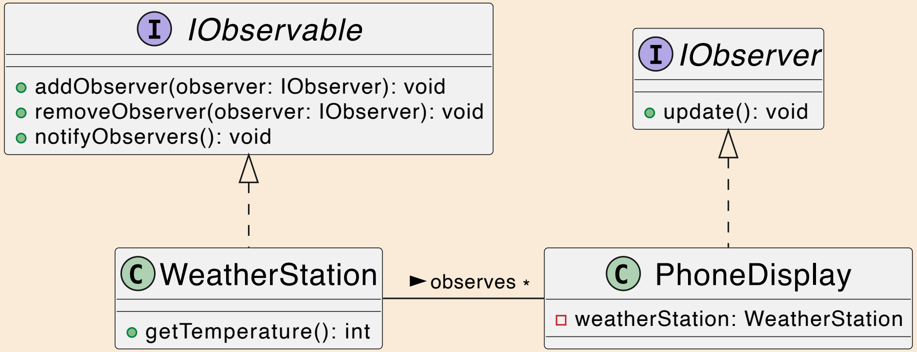
UML Diagram: Basic Structure



Real-World Example: Weather Station

- **Observable:** Weather Station measuring and updating weather data.
- **Observers:** Displays (e.g., phone display, window display) showing updated weather.

UML Diagram: Weather Station Example



Java Implementation: Interfaces

```java
public interface Observer {
  void update();
}

public interface Observable {
  void addObserver(Observer o);
  void removeObserver(Observer o);
  void notifyObservers();
}
```

Java Implementation: WeatherStation

```java
public class WeatherStation implements Observable {
  private List<Observer> observers;
  private int temperature;

  // Methods implementation...
}
```

Java Implementation: PhoneDisplay

```java
public class PhoneDisplay implements Observer {
  private WeatherStation weatherStation;

  public void update() {
    // Implementation...
  }
}
```

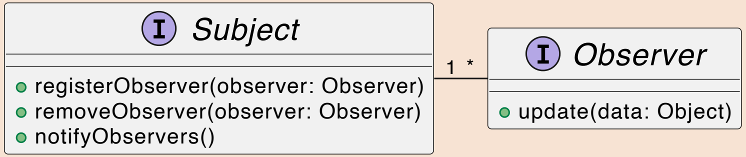
Advantages of Observer Pattern

- **Reduces Coupling:** Observers are loosely coupled with the subject.
- **Real-time Update:** Efficient update mechanism for state changes.

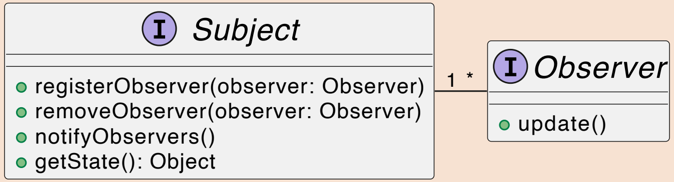
Observer Pattern: Push vs. Pull

- **Push Model:** Subject sends detailed data to observers.
- **Pull Model:** Observers request data from the subject.

UML Diagram: Push Model



UML Diagram: Pull Model



Java Implementation: Push Model

```java
public interface Observer {
  void update(Object data);
}

public class ConcreteObserver implements Observer {
  public void update(Object data) {
    // Use data directly
  }
}
```

Java Implementation: Pull Model

```java
public interface Observer {
  void update();
}

public class ConcreteObserver implements Observer {
  private ConcreteSubject subject;

  public void update() {
    Object data = subject.getState();
    // Use data
  }
}
```

Registering Observers

- Observers must register themselves to the subject.
- Allows dynamic addition and removal of observers.

Java Code: Observer Registration

```java
public class Main {
  public static void main(String[] args) {
    WeatherStation station = new WeatherStation();
    PhoneDisplay display = new PhoneDisplay(station);
    station.addObserver(display);
  }
}
```

Benefits of Observer Pattern

- **Scalability:** Easily add new observers without modifying the subject.
- **Flexibility:** Supports both push and pull data models.

Observer Pattern: Limitations

- **Potential for Memory Leaks:** Observers need to be explicitly removed.
- **Unexpected Updates:** Observers might receive updates at unpredictable times.

Summary and Conclusion

- Observer Pattern is crucial for state change notification in software design.
- Offers a robust, scalable, and flexible solution for maintaining consistency across different parts of a system.
- Suitable for various applications like UI, weather monitoring, and more.

See you in Lab!