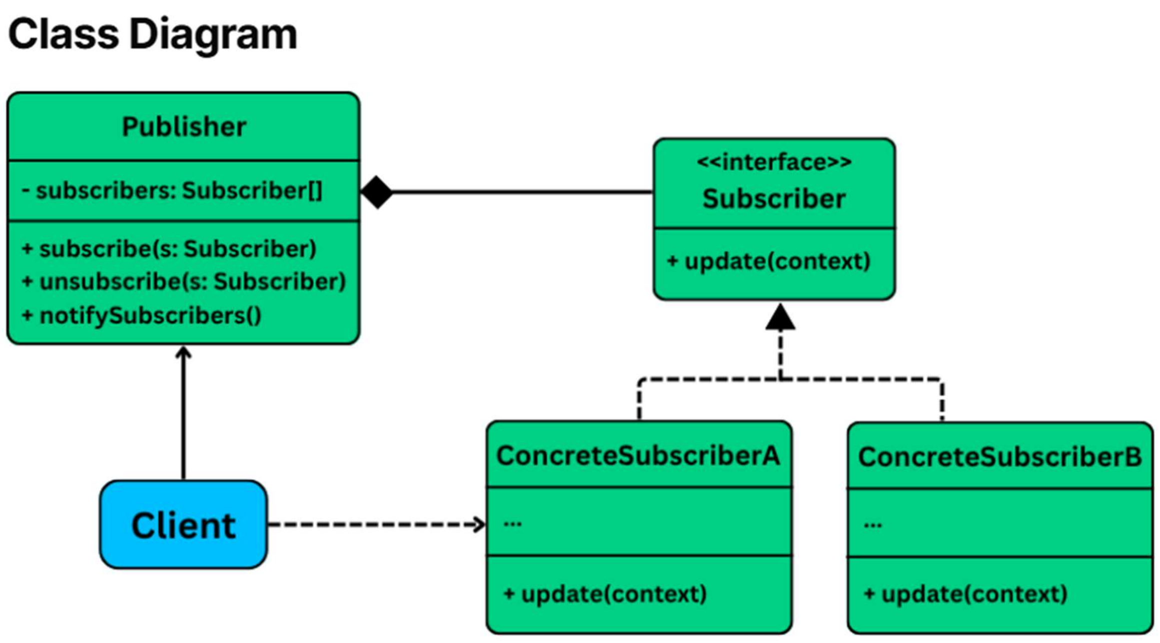
OBSERVER DESIGN PATTERN

- The **Observer Design Pattern** is a behavioral pattern that defines a **one-to-many dependency between objects** so that when one object (the subject) changes its state, all its **dependents (observers) are automatically notified and updated**.
- You have multiple parts of the system that need to react to a change in one central component.
- You want to decouple the publisher of data from the subscribers who react to it.
- You need a dynamic, event-driven communication model without hardcoding who is listening to whom.

THE OBSERVER PATTERN

- **The Observer Design Pattern** provides a clean and flexible solution to the problem of broadcasting changes from one central object (the **Subject**) to many dependent objects (the **Observers**) — all while keeping them **loosely coupled**.



1. **Observer Interface (e.g., FitnessDataObserver)**
   - Declares an update() method.
   - All modules that want to listen to fitness data changes will implement this interface.
   - Each observer defines its own logic inside update() to respond to updates.
2. **Subject Interface (e.g., FitnessDataSubject)**

   Declares methods to:

- registerObserver() – subscribe to updates
- removeObserver() – unsubscribe from updates
- notifyObservers() – notify all current observers of a change

3. **ConcreteSubject (e.g., FitnessData**)
   - Implements FitnessDataSubject.
   - Maintains an internal list of FitnessDataObserver objects.
   - When new data is pushed, it updates its internal state and calls notifyObservers() to broadcast the change.
4. **ConcreteObservers (e.g., LiveActivityDisplay)**
   - Implement the FitnessDataObserver interface.
   - When update() is called, each observer pulls relevant data from the subject and performs its own logic (e.g., update UI, log progress, send alerts).

3. IMPLEMENTING OBSERVER

- **Define the FitnessDataObserver Interface**

```
interface FitnessDataObserver {
   void update(FitnessData data);
}
```

- **Define the FitnessDataSubject Interface**

```
interface FitnessDataSubject {
   void registerObserver(FitnessDataObserver observer);
   void removeObserver(FitnessDataObserver observer);
   void notifyObservers();
}
```

- **Implement the FitnessData Class (ConcreteSubject)**

```
public class FitnessData implements FitnessDataSubject {
   private int steps;
   private int activeMinutes;
   private int calories;

   private final List<FitnessDataObserver> observers = new ArrayList<>();

   @Override
   public void registerObserver(FitnessDataObserver observer) {
      observers.add(observer);
   }

   @Override
   public void removeObserver(FitnessDataObserver observer) {
      observers.remove(observer);
```

```
    }

    @Override
    public void notifyObservers() {
        for (FitnessDataObserver observer : observers) {
            observer.update(this);
        }
    }

    public void newFitnessDataPushed(int steps, int activeMinutes, int calories) {
        this.steps = steps;
        this.activeMinutes = activeMinutes;
        this.calories = calories;

        System.out.println("\nFitnessData: New data received – Steps: " + steps +
            ", Active Minutes: " + activeMinutes + ", Calories: " + calories);

        notifyObservers();
    }

    public void dailyReset() {
        this.steps = 0;
        this.activeMinutes = 0;
        this.calories = 0;

        System.out.println("\nFitnessData: Daily reset performed.");
        notifyObservers();
    }

    // Getters
    public int getSteps() { return steps; }
    public int getActiveMinutes() { return activeMinutes; }
    public int getCalories() { return calories; }
}
```

- Implement Observer Modules

```
class LiveActivityDisplay implements FitnessDataObserver {
    @Override
    public void update(FitnessData data) {
        System.out.println("Live Display → Steps: " + data.getSteps() +
            " | Active Minutes: " + data.getActiveMinutes() +
            " | Calories: " + data.getCalories());
    }
}
```

```
class ProgressLogger implements FitnessDataObserver {
    @Override
    public void update(FitnessData data) {
        System.out.println("Logger → Saving to DB: Steps=" + data.getSteps() +
            ", ActiveMinutes=" + data.getActiveMinutes() +
            ", Calories=" + data.getCalories());
        // Simulated DB/file write...
    }
}
```

```
class GoalNotifier implements FitnessDataObserver {
    private final int stepGoal = 10000;
    private boolean goalReached = false;

    @Override
    public void update(FitnessData data) {
        if (data.getSteps() >= stepGoal && !goalReached) {
            System.out.println("Notifier → 🎉 Goal Reached! You've hit " + stepGoal + " steps!");
            goalReached = true;
        }
    }

    public void reset() {
        goalReached = false;
    }
}
```

- **Client Code**

```
public class FitnessAppObserverDemo {
    public static void main(String[] args) {
        FitnessData fitnessData = new FitnessData();

        LiveActivityDisplay display = new LiveActivityDisplay();
        ProgressLogger logger = new ProgressLogger();
        GoalNotifier notifier = new GoalNotifier();

        // Register observers
        fitnessData.registerObserver(display);
        fitnessData.registerObserver(logger);
        fitnessData.registerObserver(notifier);

        // Simulate updates
        fitnessData.newFitnessDataPushed(500, 5, 20);
        fitnessData.newFitnessDataPushed(9800, 85, 350);
        fitnessData.newFitnessDataPushed(10100, 90, 380); // Goal should trigger
        // Daily reset
        notifier.reset();
        fitnessData.dailyReset();
    }
}
```

YOUTUBE CHANNEL SUBSCRIPTION EXAMPLE

- **When New Video is uploaded to channel it should send notify to all subscriber**

```
interface ISubscriber{
	void update();
}
interface IChannel{

	void subscribe(ISubscriber subscriber);
	void unsubscribe(ISubscriber subscriber);
	void notifySubscriber();
}

class Channel implements IChannel{

	List<ISubscriber> subscriberList;
	private String name;
	private String latestVideo;



	public Channel(String name) {
		this.name = name;
		this.subscriberList = new ArrayList<>();
	}

	@Override
	public void subscribe(ISubscriber subscriber) {
		subscriberList.add(subscriber);

	}

	@Override
	public void unsubscribe(ISubscriber subscriber) {
		subscriberList.remove(subscriber);

	}

	@Override
```

```
            public void notifySubscriber() {
                        for (ISubscriber iSubscriber : subscriberList) {
                                    iSubscriber.update();
                        }

            }
            public void uploadVideo(String title) {
                        latestVideo = title;
                        System.out.println("\n[" + name + " uploaded \"" + title + "\"]");
                        notifySubscriber();
            }

            public String getVideoData() {
                        return "\nCheckout our new Video : " + latestVideo + "\n";
            }
}

//Concrete Observer: represents a subscriber to the channel
class Subscriber implements ISubscriber {
            private String name;
            private Channel channel;


            public Subscriber(String name, Channel channel) {
                        super();
                        this.name = name;
                        this.channel = channel;
            }


            @Override
            public void update() {
                        System.out.println("Hey " + name + "," + channel.getVideoData());

            }

}
Class Main{
public static void main(String[] args) {
                        Channel channel = new Channel("CoderArmy");
                        Subscriber subs1 = new Subscriber("Varun", channel);
                        Subscriber subs2 = new Subscriber("Tarun", channel);

                        channel.subscribe(subs1);
                        channel.subscribe(subs2);

                        channel.uploadVideo("Observer Pattern Tutorial");
```

```
                    // Varun unsubscribes; Tarun remains subscribed
                    channel.unsubscribe(subs1);

                    // Upload another video: only Tarun is notified
                    channel.uploadVideo("Decorator Pattern Tutorial");
          }
}
```

WHAT WE ACHIEVED

- Loose Coupling: FitnessData doesn't care who is listening — it just broadcasts
- Extensibility: Adding a new module (like WeeklySummaryGenerator) only requires implementing FitnessDataObserver — no changes to FitnessData
- Runtime Flexibility: Observers can be added/removed dynamically (e.g., based on user settings)
- Clean Separation of Concerns: Each module is responsible for its own behavior and logic