Observer Design Pattern

Observer Design Pattern

- The Observer design pattern is a behavioral pattern in software engineering that allows an object, known as the *subject*, to notify other objects, called *observers*, about changes in its state.
- This design pattern is quite powerful for building systems that need to react to certain events or changes.

One-to-Many Relationship

- **Basic Concept:**
  - The **Observer pattern** defines a one-to-many dependency between objects.
  - Whenever a change occurs in one object (the *subject*), all dependent objects (*observers*) are automatically notified and updated.
- **Components of the Observer Pattern:**
  1. **Subject**: The object that maintains a list of observers and notifies them of changes.
  2. **Observer**: The object that wants to be notified when an event occurs in the subject.
  3. **Event**: A change or action that occurs within the subject, which triggers notification to the observers.
- **Execution Sequence:**
  1. **Registration:** Observers are **registered** with the **subject** object.
  2. **Event Occurrence:** An event occurs inside the subject object.
  3. **Notification:** The subject object **notifies** all observers.
  4. **Reaction**: The observers **might react** to the event.
  5. **Deregistration**: Observers can **deregister** to stop observing.

- **Implementation:**
  - **Basic Subject Class:**

```
// Typically, there should be an interface for Subject obj
public class Subject {
    private List<Observer> observers;

    public void addObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    // Should be called every time an event occurs
    private void notifyObservers() {
        for (Observer o : observers) {
            o.update();
        }
    }
}
```

  - **Basic Observer Interface:**

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

// update() is called by the Subject object, so the Observ
// knows when an event occurred

// The code inside update() specifies how the Observer
// object responds to an event (different observers might
// react differently)
```

```
// The Observer object must register with the Subject obje
// E.g. subject.addObserver(observer);
// E.g. subject.removeObserver(observer);
```

- **Example**:
  - **Basketball Game Scenario:** In this example, we use a basketball game to demonstrate the pattern. Fans (observers) react to scores and events in a game (subject). Here's the flow:
    1. **Registering Observers**: Observers (fans) register with a subject (game). For example:

       ```
       game.addObserver(fan1);
       game.addObserver(fan2);
       ```

    2. **Event Occurrence**: An event occurs inside the subject, e.g., a team scores.

       ```
       game.scorePoints("Duke", 2);
       ```

    3. **Notifying Observers**: The subject notifies all registered observers when an event occurs.

       ```
       notifyObservers();
       ```

    4. **Observers React**: Each observer executes its `update()` method based on the event.

       ```
       fan1.update();
       fan2.update();
       ```

  - **Comprehensive Example: Basketball Game**
    - `Game` Interface:

```
public interface Game {
    void addObserver(Fan o);
    void removeObserver(Fan o);
    void notifyObservers();
    void scorePoints(String team, int points);
    void printScore();
    String whoIsWinning();
}
```

- `GameImpl` Class:

```
public class GameImpl implements Game {
    private List<Fan> fans; // Observer list encapsulated
    private String home;
    private String visitor;
    private int homeScore;
    private int visitorScore;

    public GameImpl(String visitingTeam, String homeTeam)
        fans = new ArrayList<Fan>();
        home = homeTeam;
        visitor = visitingTeam;
        homeScore = 0;
        visitorScore = 0
    }

    public void scorePoints(String team, int points) {
        if (home.equals(team)) {
            homeScore += points;
        } else if (visitor.equals(team)) {
            visitorScore += points;
        }

        // Call notifyObservers() inside the subject when
        notifyObservers();
```

```
    }

    public void printScore() {
        System.out.println(visitor + " " + visitorScore
            + " - " + homeScore + " " + home);
    }

    public String whoIsWinning() {
        if (homeScore > visitorScore) {
            return home;
        } else if (homeScore < visitorScore) {
            return visitor;
        }

        return "Tie Game";
    }

    // The subject provides methods to add and remove obse
    public void addObserver(Fan f) {
        fans.add(f);
    }

    public void removeObserver(Fan f) {
        fans.remove(f);
    }

    // notifyObservers() calls update() on each one of the
    // registered observers
    public void notifyObservers() {
        for (Fan f : fans) {
            f.update();
        }
    }
}
```

- `Fan` Interface:

```
public interface Fan {
    // Method for updating the Fan when an event occurs
    public void update();
}
```

- `UNCFan` Class:

```
public class UNCFan implements Fan {
    private Game game;

    public UNCFan(Game g) {
        game = g;
        game.addObserver(this);
    }

    public void update() {
        if (game.whoIsWinning().equals("UNC")) {
            System.out.println("UNC Fan: Go Heels!");
        }
    }
}
```

- `DukeFan` Class:

```
public class DukeFan implements Fan {
    private Game game;

    public DukeFan(Game g) {
        game = g;
        game.addObserver(this);
    }

    public void update() {
```

```
        if (game.whoIsWinning().equals("Duke")) {
            System.out.println("Duke Fan: Go Devils!");
        }
    }
}
```

- `Main` Class: (Usage)

```
public class Main {
    public static void main(String[] args) {
        Game g = new GameImpl("Duke", "UNC");
        UNCFan tarheel = new UNCFan(g);
        DukeFan devil = new DukeFan(g);
        g.scorePoints("Duke", 2); // Prints: Duke Fan: Go
        g.scorePoints("UNC", 3); // Prints: UNC Fan: Go He
        g.printScore();
    }
}
```

Many-to-Many Relationship

- In a **many-to-many** observer pattern:
  - A single observer can **observe multiple subjects.**
  - Multiple subjects can be observed by **multiple observers.**
  - Each observer **must be able to distinguish which subject** triggered the event.
- **Key Modifications Needed:**
  1. **Observer Registration**: Each observer should register with multiple subjects, and each subject should maintain its own list of observers.
  2. **Event Notification with Context**: When a subject triggers an event, it should pass itself or an event object as a parameter, allowing the observer to understand which subject triggered the notification.
- **Implementation:**

- **Updated `Subject` Class:**
  - We update the `notifyObservers` method to pass context information (the subject itself) to the observers.

```
public class Subject {
  private List<Observer> observers = new ArrayList<>();

  public void addObserver(Observer o) {
    observers.add(o);
  }

  public void removeObserver(Observer o) {
    observers.remove(o);
  }

  protected void notifyObservers() {
    for (Observer o : observers) {
        o.update(this);
    }
  }
}
```

- **Updated `Observer` Interface:**
  - We modify the `Observer` interface to include a reference to the subject that caused the notification.

```
public interface Observer {
  void update(Subject subject);
}
```

- **Example:** Let's assume we have multiple games going on at the same time.
  - **Game Interface**:

```
public interface Game {
  void addObserver(Fan o);
  void removeObserver(Fan o);
  void notifyObservers();
  void scorePoints(String team, int points);
  void printScore();
  String whoIsWinning();
}
```

- **Game Implementation**:

```
public class GameImpl implements Game {
    private List<Fan> fans;
    private String home;
    private String visitor;
    private int homeScore;
    private int visitorScore;

    public GameImpl(String visitingTeam, String homeTeam)
{
        fans = new ArrayList<>();
        home = homeTeam;
        visitor = visitingTeam;
        homeScore = 0;
        visitorScore = 0;
    }

    public void addObserver(Fan f) {
        fans.add(f);
    }

    public void removeObserver(Fan f) {
        fans.remove(f);
    }
```

```
    public void scorePoints(String team, int points) {
        if (home.equals(team)) {
            homeScore += points;
        } else if (visitor.equals(team)) {
            visitorScore += points;
        }
        notifyObservers();
    }

    public void notifyObservers() {
        for (Fan f : fans) {
            f.update(this);  // Pass this game instance to
the fan
        }
    }

    public void printScore() {
        System.out.println(visitor + " " + visitorScore +
" - " + homeScore + " " + home);
    }

    public String whoIsWinning() {
        if (homeScore > visitorScore) {
            return home;
        } else if (homeScore < visitorScore) {
            return visitor;
        }
        return "Tie Game";
    }
}
```

- **Fan Interface**:

```
public interface Fan extends Observer {
    void update(Game g);
```

```
}
```

- **UNCFan and DukeFan Classes**: These classes need to handle multiple games, so they should accept the game instance in the `update` method.

```
public class UNCFan implements Fan {
    public void update(Game g) {
        if (g.whoIsWinning().equals("UNC")) {
            System.out.println("UNC Fan: Go Heels!");
        }
    }
}

public class DukeFan implements Fan {
    public void update(Game g) {
        if (g.whoIsWinning().equals("Duke")) {
            System.out.println("Duke Fan: Go Devils!");
        }
    }
}
```

- **Main Class for Many-to-Many Example:**

```
public class Main {
    public static void main(String[] args) {
        Game[] games = new Game[3];
        games[0] = new GameImpl("Duke", "UNC");
        games[1] = new GameImpl("UNC", "NC State");
        games[2] = new GameImpl("Duke", "Georgetown");

        UNCFan tarheel = new UNCFan();
        DukeFan devil = new DukeFan();

        // Register the fans to multiple games
        for (Game g : games) {
```

```
            g.addObserver(tarheel);
            g.addObserver(devil);
        }

        // Trigger some events in different games
        games[0].scorePoints("UNC", 2);
        games[1].scorePoints("NC State", 2);
        games[2].scorePoints("Duke", 3);
    }
}
```

- **Explanation of Changes:**
    1. **Passing Context**: In the example, each `Game` object is passed into the `update` method, so the fans can distinguish which game generated the event. This is key in a many-to-many relationship.
    2. **Observer Registration**: We register each observer (fan) with all games. This sets up a many-to-many relationship between games and fans.

Passing Event Contexts to Observers

- This extension of the **Observer design pattern** involves passing **event context** to the observers, enabling more complex and meaningful reactions to events.
- This pattern modification introduces an **event object** that holds contextual information about what happened, allowing observers to act based on the event details.
- **Key Changes:**
    1. **Event Object**: Encapsulates specific details about an event that occurred in the subject (like a game).
    2. **Enhanced Notification Method**: The `notifyObservers` method now passes an event object (`Event e`) to the observers, providing additional context.
    3. **Observer Interface**: Modified to receive the subject and event objects so it can react based on the source and nature of the event.

- **Implementation:**
  - `Event` Object: An event object is used to encapsulate information about the change or occurrence. It allows observers to react more specifically based on the event's details.
  - Updated `Observer` Interface: The observer interface is modified to accept both the subject and the event. This allows the observer to know:
    - Which object caused the event.
    - What actually happened (through the event object).
  - **Subject Class with Event Context:**

```
public class Subject {
  private List<Observer> observers = new ArrayList<>();

  public void addObserver(Observer o) {
    observers.add(o);
  }

  public void removeObserver(Observer o) {
      observers.remove(o);
  }

  protected void notifyObservers(Event e) {
    for (Observer o : observers) {
       o.update(this, e); // Pass the subject (this) and the
    }
  }
}
```

  - **Observer Interface:**

```
public interface Observer {
      void update(Subject s, Event e);
}
```

- **Game Example:**
  - **Game Interface**:

```
public interface Game {
    void scorePoints(String team, int points);
    String whoIsWinning();
}
```

  - **Game Implementation:**

```
public class GameImpl implements Game {
    private List<Fan> fans;
    private String home;
    private String visitor;
    private int homeScore;
    private int visitorScore;

    public GameImpl(String visitingTeam, String homeTeam) {
        fans = new ArrayList<>();
        home = homeTeam;
        visitor = visitingTeam;
        homeScore = 0;
        visitorScore = 0;
    }

    public void addObserver(Fan f) {
        fans.add(f);
    }

    public void removeObserver(Fan f) {
        fans.remove(f);
    }

    public void scorePoints(String team, int points) {
        String winningBefore = whoIsWinning();
```

```
        if (home.equals(team)) {
            homeScore += points;
        } else if (visitor.equals(team)) {
            visitorScore += points;
        }
        GameEvent e;
        if (winningBefore.equals("Tie Game")) {
            e = new GameEventImpl("Lead Change", team);
        } else if (Math.abs(homeScore - visitorScore) > 10) {
            e = new GameEventImpl("Big Lead", team);
        } else {
            e = new GameEventImpl("Score Change", team);
        }
        notifyObservers(e);
    }

    public void notifyObservers(GameEvent e) {
        for (Fan f : fans) {
            f.update(this, e); // Notify fans with event cont
        }
    }

    public String whoIsWinning() {
        if (homeScore > visitorScore) {
            return home;
        } else if (homeScore < visitorScore) {
            return visitor;
        }
        return "Tie Game";
    }
}
```

- **GameEvent Interface:**

```
public interface GameEvent {
    String getType();
    String getWhoScored();
}
```

- **GameEvent Implementation**:

```
public class GameEventImpl implements GameEvent {
    private String type;
    private String whoScored;

    public GameEventImpl(String type, String whoScored) {
        this.type = type;
        this.whoScored = whoScored;
    }

    public String getType() {
        return type;
    }

    public String getWhoScored() {
        return whoScored;
    }
}
```

- **Fan Interface:**

```
public interface Fan extends Observer {
    void update(Game g, GameEvent e);
}
```

- **UNCFan and DukeFan Classes**:

```
public class UNCFan implements Fan {
    public void update(Game g, GameEvent e) {
        if (e.getType().equals("Lead Change") && e.getWhoScor
            System.out.println("UNC Fan: Go Heels, we took th
        } else if (e.getType().equals("Big Lead") && e.getWho
            System.out.println("UNC Fan: Go Heels, we have a
        } else if (e.getType().equals("Score Change") && e.ge
            System.out.println("UNC Fan: Nice, more points fo
        }
    }
}

public class DukeFan implements Fan {
    public void update(Game g, GameEvent e) {
        if (e.getType().equals("Lead Change") && e.getWhoScor
            System.out.println("Duke Fan: Go Devils, we took
        } else if (e.getType().equals("Big Lead") && e.getWho
            System.out.println("Duke Fan: Go Devils, we have
        } else if (e.getType().equals("Score Change") && e.ge
            System.out.println("Duke Fan: Nice, more points t
        }
    }
}
```

- **Explanation:**
    1. **Context-Aware Notification**: The observers (fans) now receive not only a reference to the subject (`Game g`) but also the specific event (`GameEvent e`). This allows them to react differently based on the event type and the details within the event.
    2. **Distinguishing Events**: Fans can distinguish between different types of events (like "Lead Change" or "Big Lead") and react accordingly.
    3. **GameEvent Encapsulation**: The `GameEventImpl` class encapsulates important information like the type of event and the team that scored. This

design is extensible, allowing you to add more event types or information if needed.

Functional Programming

- Functional Programming (FP) is a paradigm where programs are constructed using pure functions, avoiding shared state, and mutable data. Here are its primary characteristics:
  - **Immutability**: Data cannot be modified after it's created.
  - **First-Class and Higher-Order Functions**: Functions are treated as values and can be passed as arguments, returned from other functions, or assigned to variables.
  - **No Side Effects**: Pure functions don't modify any external state or variables. They always return the same result for the same input.
  - **Declarative Style**: You describe *what* to do rather than *how* to do it.
- In functional programming, you focus on functions as the primary unit of code organization. Functions can be passed around, returned from other functions, and combined to build more complex logic.
- Java, traditionally an Object-Oriented language, began incorporating functional programming constructs to make it more flexible and modern. Java introduced two major features for this purpose:
  1. **Anonymous Classes**: Letting you define classes without giving them a formal name.
     - Example:

```
// Call new <interface_name>
Fan tarheel = new Fan() {
    @Override
    public void update(Game g) {
        if (g.whoIsWinning().equals("UNC")) {
            System.out.println("UNC Fan: Go Heels!");
        }
    }
```

```
};

// Example usage: g.addObserver(tarheel);
```

2. **Lambda Expressions**: A concise way to represent single-method interfaces as inline code.
    - Example:

```
// No method name (header), just method body
Fan tarheel = (Game g) -> {
    if (g.whoIsWinning().equals("UNC")) {
        System.out.println("UNC Fan: Go Heels!");
    }
};

// Example usage: g.addObserver(tarheel);
```

    - **Note**: These are only applicable and usable when dealing with interfaces with a single method.

- **Using Switch Statements in Functional Style:**
    - Java also added enhancements to `switch` statements to make them look more like expressions (inspired by functional programming).
    - However, *this is unrelated to functional programming.*
    - Example:

```
switch (g.whoIsWinning()) {
    case "UNC" -> System.out.println("UNC Fan: GO HEELS!");
    default -> System.out.println("UNC Fan: Good job, Caroli
}
```