Design Patterns in Java

Object-Oriented Programming Class

December 16, 2024

Singleton Pattern

Purpose: Ensures a class has only one instance and provides a global point of access to it.

Example in Java:

```
1  // Singleton class
2  class Singleton {
3      private static Singleton instance;
4
5      private Singleton() {}
6
7      public static Singleton getInstance() {
8          if (instance == null) {
9              instance = new Singleton();
10         }
11         return instance;
12     }
13
14     public void showMessage() {
15         System.out.println("Hello from Singleton!");
16     }
17 }
18
```

Adapter Pattern I

Purpose: Allows classes with incompatible interfaces to work together.

Typical Example: Converting an existing interface into one expected by the client.

Example in Java:

```
1  // Target interface
2  interface Target {
3      void request();
4  }
5
6  // Existing class with an incompatible interface
7  class Adaptee {
8      void specificRequest() {
9          System.out.println("Called specific request
               method.");
10     }
11 }
12
13 // Adapter converting Adaptee to Target
```

Adapter Pattern II

```
14  class Adapter implements Target {
15      private Adaptee adaptee;
16
17      Adapter(Adaptee adaptee) {
18          this.adaptee = adaptee;
19      }
20
21      @Override
22      public void request() {
23          adaptee.specificRequest();
24      }
25  }
26
27  // Usage
28  public class Main {
29      public static void main(String[] args) {
30          Adaptee adaptee = new Adaptee();
31          Target adapter = new Adapter(adaptee);
32          adapter.request(); // Output: Called specific
                request method.
```

Adapter Pattern III

```
33          }
34      }
```

Decorator Pattern I

Purpose: Dynamically add responsibilities to objects without modifying their code.
Typical Example: Wrapping objects to add additional functionalities.
Example in Java:

```
1  // Base component
2  interface Component {
3      void operation();
4  }
5
6  // Concrete implementation
7  class ConcreteComponent implements Component {
8      public void operation() {
9          System.out.println("Basic operation.");
10     }
11 }
12
13 // Base decorator
14 abstract class Decorator implements Component {
```

Decorator Pattern II

```
15      protected Component component;
16
17      Decorator(Component component) {
18          this.component = component;
19      }
20
21      public void operation() {
22          component.operation();
23      }
24  }
25
26  // Concrete decorator
27  class ConcreteDecorator extends Decorator {
28      ConcreteDecorator(Component component) {
29          super(component);
30      }
31
32      @Override
33      public void operation() {
34          super.operation();
```

Decorator Pattern III

```
35            System.out.println("Additional functionality.");
36        }
37    }
38
39    // Usage
40    public class Main {
41        public static void main(String[] args) {
42            Component base = new ConcreteComponent();
43            Component decorated = new ConcreteDecorator(base);
44            decorated.operation();
45            // Output: Basic operation.
46            //         Additional functionality.
47        }
48    }
```

Strategy Pattern I

Purpose: Allows defining a family of algorithms, encapsulating them, and making them interchangeable.
Typical Example: Changing the behavior of an object at runtime.
Example in Java:

```
1  // Strategy
2  interface Strategy {
3      int execute(int a, int b);
4  }
5
6  // Concrete implementations
7  class Addition implements Strategy {
8      public int execute(int a, int b) {
9          return a + b;
10     }
11 }
12
13 class Multiplication implements Strategy {
14     public int execute(int a, int b) {
15         return a * b;
```

Strategy Pattern II

```
16      }
17  }
18
19  // Context using a strategy
20  class Context {
21      private Strategy strategy;
22
23      public void setStrategy(Strategy strategy) {
24          this.strategy = strategy;
25      }
26
27      public int executeStrategy(int a, int b) {
28          return strategy.execute(a, b);
29      }
30  }
31
32  // Usage
33  public class Main {
34      public static void main(String[] args) {
35          Context context = new Context();
```

Strategy Pattern III

```
36
37              context.setStrategy(new Addition());
38              System.out.println("Sum: " + context.
                    executeStrategy(3, 4));
39
40              context.setStrategy(new Multiplication());
41              System.out.println("Multiplication: " +
                    context.executeStrategy(3, 4));
42          }
43      }
```

Observer Pattern I

Purpose: Defines a one-to-many dependency between objects so that when one changes state, all its dependents are notified automatically.
Typical Example: Event systems, such as a notification system.
Example in Java:

```
1  // Observable subject
2  import java.util.ArrayList;
3  import java.util.List;
4
5  class Subject {
6      private List<Observer> observers = new ArrayList
           <>();
7      private int state;
8
9      public void addObserver(Observer observer) {
10         observers.add(observer);
11     }
12
13     public void setState(int state) {
```

Observer Pattern II

```
14              this.state = state;
15              notifyObservers();
16          }
17
18          private void notifyObservers() {
19              for (Observer observer : observers) {
20                  observer.update(state);
21              }
22          }
23      }
24
25      // Observer
26      interface Observer {
27          void update(int state);
28      }
29
30      class ConcreteObserver implements Observer {
31          private String name;
32
33          ConcreteObserver(String name) {
```

Observer Pattern III

```
34              this.name = name;
35          }
36
37          public void update(int state) {
38              System.out.println(name + " received update: "
                    + state);
39          }
40      }
41
42      // Usage
43      public class Main {
44          public static void main(String[] args) {
45              Subject subject = new Subject();
46
47              Observer observer1 = new ConcreteObserver("
                    Observer 1");
48              Observer observer2 = new ConcreteObserver("
                    Observer 2");
49
50              subject.addObserver(observer1);
```

Observer Pattern IV

```
51              subject.addObserver(observer2);
52
53              subject.setState(5);
54              subject.setState(10);
55          }
56      }
```