Design Patterns

Josh Wilcox (jw14g24)

May 10, 2025

Table of Contents

What are Design Patterns

What are Design Patterns

- They are reusable, general solutions to common software design problems
- Template for solving a problem that can be deployed in many situations
- Can be viewed as **best practices**
- They typically show ideal relationships between objects without specification of the final objects involved

---

Groups of Design Patterns

- **Creational Patterns**
  - Deal with object creation mechanisms that create objects in a manner suitable to the situation
  - **Example** - The *singleton* design pattern that ensures only one instance of a class exists
- **Structural Patterns**
  - Realize relationship among entities and focus on combinations and relations between objects
  - **Example** - The *composite* pattern that creates a tree structure of objects that all share the same interface
- **Behavioural Patterns**
  - Identify commun communication patterns among objects, increasing flexibility in carrying out communication
  - **Example** - The *iterator* pattern that can sequentially access elements of an aggregate object without exposing underlying representation
  - **Example** - The *functional interface* design pattern

Why use Design Patterns

- **Reusability** - Reduces code duplication
  - Provides tested, proven development paradigms
  - Allows reuse of common problem-solving approaches
- **Maintainability** - Improves code organisation
  - Standardizes code structure
  - Makes code more readable and self-documenting
- **Scalability** - Systems using design patterns are easily expandable
  - Modular structure allows easy addition of new features
  - Patterns are designed to accommodate growth
- **Flexibility** - Encourages **loose coupling**
  - Components can be modified independently
  - Reduces impact of changes across the system
- **Encapsulation** - Isolates frequently changing parts of code
  - Hides implementation details from clients
  - Makes system more robust to internal changes

Functional Interface Pattern

What are Functional Interfaces

- In the context of Java, Functional Interfaces at *function objects*
  - These means they **only contain *one* function**
- In Java, Functional Interfaces implement and interface that contain a single function

The Need for Functional Interfaces

- It is common to vary how an algorithm is performed without changing the **method that implements an algorithm**
- This can be acheived by passing the method the means of performing the operation

- **Decouple the application from the implementation**
- **Encapsulate the implementation**
- Functional Interfaces achieve this by:
  - Allowing functions to be passed as parameters
  - Enabling runtime selection of which function to execute
  - Separating the algorithm from the specific operation it performs

How Functional Interfaces are used in java

- The solution for using Functional Interfaces involves:
  - Using/defining an interface with the desired function header
  - Defining a class that implements the interface
  - Creating an instance of this class when needed
  - Passing the instance to methods that need it
- For comparing objects, Java provides the built-in `Comparator<T>` interface
  - Used extensively in Arrays and Collections methods
  - Simplifies object comparison operations

While this approach may seem cumbersome, it provides a structured way to implement function objects in Java's object-oriented paradigm.

- Functional interfaces are typically implemented by **lambda expressions**

Builtin Functional Interfaces

- `Function<T,R>` - Takes argument `T` and produces result `R`
- `Predicate<T>` - Returns a boolean value based on `T`
- `Consumer<T>` - Takes argument `T` and manipulates without returning
- `Supplier<T>` - **Does not take input** - and returns type `T` objects
- `BiFunction<T,U,R>` - Takes arguments `T` and `R` and returns `R`
- `BiPredicate<T,U>` - Returns a boolean value based on `T` and `U`
- `UnaryOperator<T>` - Equivalent to `Function<T,T>`
- `BinaryOperator<T>` - Equivalent to `BiFunction<T,T,T>`

Iterator Pattern

What are Iterators

- Automated means of traversing a collection of objects
- It decouples algorithms from containers - meaning we do not need to know the underlying scheme of an object to do processes such as sorting if we setup an Iterator correctly
- You can iterate across a data structure without knowing implementation detail

Iterators in Java

- Iterators in Java are *interfaces* with the following required methods:
  - `boolean hasNext();`
    - Are there any objects left to iterate over
  - `E next();`
    - Gets the next object
- Objects that implement the `Iterable` interface contain their own iterator which defined how they can be searched through
  - An `ArrayList` is an example of such an object
  - The `Iterable` interface for the use of `for each` looping

Custom Iterator Example

```java
public class CustomList<T> implements Iterable<T> {
    private T[] items;
    private int size;

    // Constructor and other methods...

    @Override
    public Iterator<T> iterator() {
        return new CustomIterator();
    }

    private class CustomIterator implements Iterator<T> {
        private int currentIndex = 0;

        @Override
        public boolean hasNext() {
            return currentIndex < size;
        }

        @Override
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return items[currentIndex++];
        }
    }
}
```

Composite Pattern

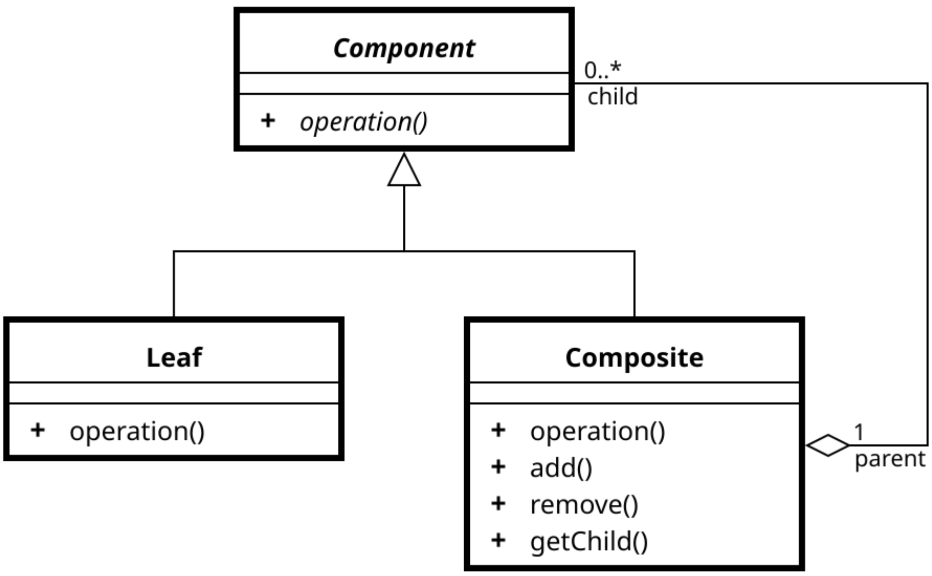
What is the Composite Patterns

- Key Idea: Unified interfaces for both **individual** and **collections** of those individual objects
- Allows for treating both collections and the individual objects that make up those collections in the same way
  - For example, the renaming of a folder in finder is done in exactly the same way as renaming a file, but a folder is a collection of files
- Consists of two types of objects:
  - **Leaf Objects** - Represent the simple elements in the system
  - **Composite Objects** - Groups or collections that are in the hierarchy above the leaf objects

Example Use cases

- File System
  - **Leaf**: File
  - **Composite**: Folder or Directory
- Graphics
  - **Leaf**: Triangle
  - **Composite**: Render made out of small triangles
- GUI
  - **Leaf**: A button
  - **Composite**: A gridpane
- Organisation
  - **Leaf**: Single Employee
  - **Composite**: A team

UML



- A leaf **is a** component
- A composite **is a** component
- A composite **has many** components
  - These components can either be leaves or more composites

Singleton Pattern

What is a Singleton Pattern

- They restrict the instantiation of a class to a **singular instance**
- Used when exactly one object is needed to coordinate actions across a system
- Affords the following:
  - Ensuring classes only have one instance
  - Access to that instance is easy
  - Instantiation of such instance is controlled

Implementation Example

```java
public class Singleton {
    private static Singleton instance;
    private Singleton() {} // private constructor

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

Enums as Singletons

- Java Enums naturally follow the singleton pattern
  - Each enum constant is instantiated only once
  - They are thread-safe by default
  - Cannot be instantiated using constructors
- Example:

```java
public enum SingletonEnum {
    INSTANCE;

    public void doSomething() {
        // singleton behavior
    }
}
```

> Using enums for singletons is often considered the best practice in Java as it provides serialization safety and thread-safety guarantees.

Observer Pattern

What is the Observer Pattern

- Defines a one-to-many dependency between objects
- When one object (the subject) changes state, all its dependents (observers) are notified automatically
- Promotes loose coupling between subjects and observers
  - Subject only knows that observers implement a specific interface
  - Observers can be added/removed without modifying the subject

Observer Pattern in JavaFX

- JavaFX extensively uses the Observer pattern through:
  - **Properties** - Observable values that notify listeners of changes
  - **Bindings** - Create dependencies between properties
  - **Event Handlers** - React to user interface events
- Common JavaFX examples:
  - Button clicks triggering actions
  - Text fields updating labels
  - Property bindings for automatic UI updates

JavaFX Property Example

```java
// JavaFX implementation
StringProperty text = new SimpleStringProperty();
Label label = new Label();
label.textProperty().bind(text); // Observer pattern
text.set("New Value"); // Automatically updates label
```

- Key benefits in GUI:
  - Automatic UI updates
  - Decoupled components
  - Event-driven programming
  - Reactive interfaces