Design Pattern

Josh Wilcox (jw14g24@soton.ac.uk)

March 11, 2025

Contents

1 What are Design Patterns

- A reusable **general solution** to a *common* software design problem

1.1 Why use Design Patterns

- **Reusability** - Reduces code duplication by using standardised solutions to common problems
- **Maintainability** - Improves code organisation
  - Easier to modify and extend
- **Scalability** - Allows systems to grow and adapt efficiency
- **Flexibility** - Encourages *loose coupling*, allowing components to be interchangable
- **Code Readability** - Makes code more understandable to people familiar with design patterns
- **Enapsulation** - Isolates frequently changing parts of the code
  - Minimises the impact of change on the entire system

2 Patterns

2.1 Creational Design Patterns

- **Singleton**
  - Guarantees a single instance of a class, offering a global access point.
  - Ideal for managing shared resources such as configurations or connection pools.
- **Factory Method**
  - Delegates object creation to subclasses through an overridable method.
  - Improves flexibility by abstracting the instantiation process.
- **Abstract Factory**
  - Creates families of related objects without specifying their concrete classes.
  - Ensures consistency among products and simplifies switching between families.
- **Builder**
  - Constructs complex objects step by step, separating construction from representation.
  - Allows different configurations for the final object, boosting maintainability.
- **Prototype**
  - Generates new objects by copying an existing instance.
  - Efficient when object creation is costly and similarities exist across objects.

2.2 Structural Design Patterns

- **Adapter**
  - Converts one interface into another to allow incompatible systems to work together.
  - Facilitates integration with legacy systems or third-party libraries.
- **Bridge**
  - Separates an abstraction from its implementation, enabling them to evolve independently.
  - Enhances scalability by accommodating a range of implementations without modifying high-level abstractions.

- **Composite**
  - Organizes objects into tree structures for part-whole hierarchies.
  - Simplifies client code by treating individual objects and composites uniformly.
- **Decorator**
  - Dynamically attaches additional responsibilities to an object.
  - Adheres to the open-closed principle by extending functionality without modifying existing code.
- **Facade**
  - Offers a simplified interface to a complex subsystem.
  - Decouples client interactions from detailed subsystem operations, improving clarity.
- **Flyweight**
  - Reduces memory usage by sharing common parts of object state.
  - Suitable for managing large numbers of fine-grained objects efficiently.
- **Proxy**
  - Controls access to a target object, often adding a layer of security or managing resources.
  - Useful for lazy initialization, logging requests, or enforcing access policies.

2.3 Behavioural Design Patterns

- **Functor**
  - Treats functions as objects, allowing them to be passed and manipulated like data.
  - Enables advanced functional programming techniques and composition of behaviors.
- **Iterator**
  - Provides sequential access to elements in a collection without exposing underlying structure.
  - Simplifies traversal operations and supports multiple concurrent iterations.
- **Chain of Responsibility**
  - Passes requests along a chain of handlers until one processes the request.
  - Decouples senders and receivers, allowing dynamic configuration of request handling.
- **Command**
  - Encapsulates a request as an object, allowing parameterization and queueing of operations.
  - Supports undoable operations and transaction-like behavior in applications.
- **Interpreter**
  - Implements a language interpreter by representing grammar rules as classes.
  - Useful for parsing domain-specific languages and structured input formats.
- **Mediator**
  - Centralizes communication between objects through a mediator object.
  - Reduces coupling between components and simplifies interaction management.
- **Memento**
  - Captures and preserves an object's internal state without violating encapsulation.
  - Enables implementation of undo mechanisms and history tracking features.
- **Observer**

  - Defines a one-to-many dependency where changes in one object automatically notify others.
  - Supports event handling systems and reactive programming models.
- **State**
  - Allows an object to change its behavior when its internal state changes.
  - Simplifies complex conditional logic by encapsulating state-specific behavior in separate classes.
- **Strategy**
  - Defines interchangeable algorithms that can be selected at runtime.
  - Promotes flexibility by isolating algorithm implementations from the code that uses them.
- **Template Method**
  - Defines an algorithm's structure while allowing subclasses to override specific steps.
  - Maintains consistency across implementations while enabling customization.
- **Visitor**
  - Separates algorithms from the objects they operate on, allowing new operations without modifying classes.
  - Facilitates adding functionality to existing class hierarchies while maintaining encapsulation.

3 The Functor Pattern

- Functors are **Function Objects**
  - These are objects that only contain a *single function*
- Functors implement an interface **containing a single function**
- A way of replacing function pointers from `C/C++`

3.1 The need for Functors

- We may want to vary how an operation is performed without changing the method that implements an algorithm
- **Decouples** the application from the implementation
  - Encapsulates the implementation

3.2 Functors in Java

a) Define an interface with the function headerr we want
b) Define a class that implements the interface
   - Could be a builtin interface or could be our own
c) Create an instance of this class when needed
d) Pass the instance of the method that needs it

4 The iterator Pattern

- The Iterator pattern provides a way to **access elements** of a collection *sequentially* without exposing the underlying representation
- Separates the traversal of a collection from its implementation
- Enables multiple traversals of the same collection simultaneously

4.1 The need for Iterators

- Collections can have different internal structures (arrays, linked lists, trees, etc.)
- We want to **standardize** how we access elements regardless of the collection type
- Provides a *uniform interface* for traversing different collections
- Hides implementation details of the collection
- Allows for different traversal strategies (forward, backward, filtered, etc.)

4.2 Iterators in Java

a) Java provides the `Iterator` interface in the `java.util` package
b) Key methods:
   - `hasNext()` - Returns whether there are more elements
   - `next()` - Returns the next element
   - `remove()` - Removes the last element returned (optional operation)
c) Collections implement the `Iterable` interface to support the for-each loop
d) Custom collections can create their own iterators by implementing these interfaces
e) The `for-each` loop in Java uses iterators behind the scenes

5 Composite Pattern

- Treats individual objects and compositions of objects uniformly
- Allows the composition of objects into *tree-like* structures to represent part-whole hierarchies
- Enables working with indiviual objects and groups of objects in the same way
- Useful when we ned to work with hierarchical data

5.1 Understanding the Composite Pattern

- The Composite pattern is a **structural design pattern** that lets you:
  - Compose objects into tree structures
  - Represent part-whole hierarchies
  - Treat individual objects and compositions of objects uniformly
- The pattern consists of three key components:
  - **Component** - The interface or abstract class defining operations common to all objects
  - **Leaf** - Simple individual objects that implement the Component interface
  - **Composite** - Complex objects containing child components (both Leaf objects and other Composites)
- Enables *recursive composition* where clients can treat both simple and complex elements identically

5.2 Implementation in Java

- A typical implementation includes a common interface with operations for both simple and composite objects

```java
public interface FileSystemComponent {
    void display(String indent);
}
```

```java

public class File implements FileSystemComponent {
    private String name;

    public File(String name) {
        this.name = name;
    }

    @Override
    public void display(String indent) {
        System.out.println(indent + "- " + name);
    }
}

public class Folder implements FileSystemComponent {
    private String name;
    private List<FileSystemComponent> children = new ArrayList<>();

    public Folder(String name) {
        this.name = name;
    }

    public void add(FileSystemComponent component) {
        children.add(component);
    }

    public void remove(FileSystemComponent component) {
        children.remove(component);
    }

    @Override
    public void display(String indent) {
        System.out.println(indent + "+ " + name);
        for (FileSystemComponent component : children) {
            component.display(indent + "  ");
        }
    }
}
```

- Note the key aspects of this implementation:
  - **Common interface** (`FileSystemComponent`) shared by both `File` (leaf) and `Folder` (composite)
  - Both classes implement the same `display()` method
  - `Folder` class contains methods to manage children that are *not* in the interface
  - Client code can work with the base interface, unaware of whether it's dealing with a leaf or composite

5.3 Usage Example

```java
public class FileSystemExplorer {
    public static void main(String[] args) {
        // Create files (leaf objects)
        FileSystemComponent file1 = new File("Document.txt");
        FileSystemComponent file2 = new File("Picture.jpg");
        FileSystemComponent file3 = new File("Music.mp3");

        // Create folders (composite objects)
        Folder root = new Folder("Root");
        Folder documents = new Folder("Documents");
        Folder media = new Folder("Media");

        // Build the structure
        documents.add(file1);
        media.add(file2);
        media.add(file3);
        root.add(documents);
        root.add(media);

        // Display the entire structure using a single method
        root.display("");
    }
}
```

5.4 Benefits and Applications

- **Decoupling** - Client code is decoupled from specific component classes
- **Simplification** - Work with complex hierarchies through a simple, uniform interface
- **Extensibility** - Add new component types without changing existing code
- Common applications include:
  - **File systems** - Files and directories form natural hierarchies
  - **GUI components** - Containers (panels, windows) and widgets (buttons, text fields)
  - **Graphics systems** - Complex shapes composed of simpler shapes
  - **Organizational structures** - Employees, teams, departments, divisions
  - **Menu systems** - Menus containing submenus and menu items

5.5 Considerations and Trade-offs

- **Interface bloat** - The Component interface may include methods that don't make sense for Leaf objects
  - Some implementations use `default` methods or throw `UnsupportedOperationException`
- **Type safety** - Generic typing can help ensure appropriate children are added
- **Performance** - Deep hierarchies might impact performance for operations that traverse the entire structure
- **Memory usage** - References between parents and children can become complex