Iterator Pattern - Simplifying Traversal

What is the Iterator Pattern?

The **Iterator Pattern** is a behavioral design pattern that provides a way to sequentially access elements of a collection without exposing its underlying representation. It decouples the client from the collection's structure, making it easier to iterate over elements in a consistent way.

Key Components of the Iterator Pattern

1. **Iterator Interface**:
   - Defines methods like `hasNext()` and `next()` to traverse the collection.
2. **Concrete Iterator**:
   - Implements the Iterator interface and tracks the current position.
3. **Aggregate (Collection) Interface**:
   - Defines a method to create an iterator (e.g., `createIterator()`).
4. **Concrete Aggregate (Collection)**:
   - Implements the Aggregate interface and provides an iterator for its elements.
5. **Client**:
   - Uses the iterator to traverse the collection without knowing its internal details.

Why Use the Iterator Pattern?

- **Encapsulation**: Hides the internal structure of collections.
- **Consistency**: Provides a uniform way to traverse different types of collections.
- **Flexibility**: Allows multiple iterators to operate on the same collection independently.

How It Works

1. Create an **Iterator Interface** with traversal methods.
2. Implement a **Concrete Iterator** for the collection.
3. Define an **Aggregate Interface** with a method to return an iterator.
4. Implement the **Concrete Aggregate** that creates and returns its iterator.
5. The **Client** uses the iterator to access elements.

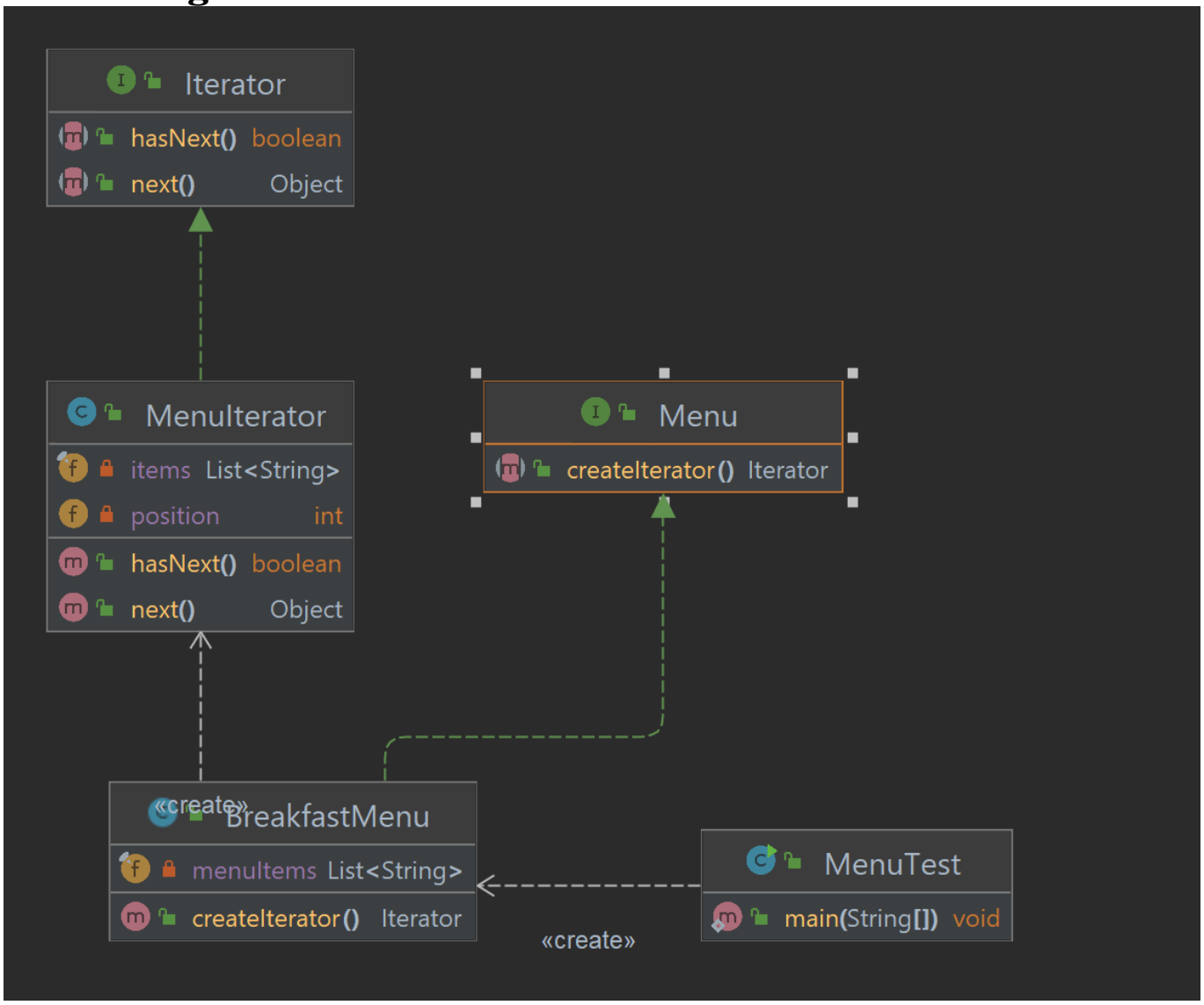
Project Structure

```
src/main/java/com/headfirst/chapter9/iterator_compositePattern/
iterator
├── menu/
│   ├── Iterator.java                # Iterator Interface
│   ├── Menu.java                    # Aggregate Interface
│   ├── MenuIterator.java            # Concrete Iterator
│   ├── BreakfastMenu.java           # Concrete Aggregate
│   └── MenuTest.java                # Client
└── filesystem/
    ├── FileSystemIterator.java      # Real-world Iterator
    └── FileSystemTest.java          # Real-world Client
```

Example: Menu Iterator

Consider a restaurant menu system where we need to traverse multiple menu collections (e.g., Breakfast Menu, Lunch Menu) without exposing their internal implementations.

Class Diagram



Example Code

Iterator Interface

```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

Concrete Iterator

```
import java.util.List;

public class MenuIterator implements Iterator {
    private final List<String> items;
    private int position = 0;

    public MenuIterator(List<String> items) {
        this.items = items;
    }
    @Override
    public boolean hasNext() {
        return position < items.size();
    }

    @Override
    public Object next() {
        if (!hasNext()) {
            throw new IllegalStateException("No more items");
        }
        return items.get(position++);
    }
}
```

Aggregate Interface

```
public interface Menu {
    Iterator createIterator();
}
```

Concrete Aggregate

```
import java.util.List;

public class BreakfastMenu implements Menu {
    private final List<String> menuItems;
```

```
    public BreakfastMenu(List<String> menuItems) {
        this.menuItems = menuItems;
    }

    @Override
    public Iterator createIterator() {
        return new MenuIterator(menuItems);
    }
}
```

Client

```
import java.util.Arrays;

public class MenuTest {
    public static void main(String[] args) {
        Menu breakfastMenu = new
         BreakfastMenu(Arrays.asList("Pancakes", "Waffles",
         "Omelette"));
         Iterator iterator = breakfastMenu.createIterator();

         System.out.println("Breakfast Menu:");
         while (iterator.hasNext()) {
             System.out.println("- " + iterator.next());
         }
    }
}
```

Real-World Applications

Example: File System Traversal

In a file system, an iterator can be used to traverse files and directories without exposing the underlying structure.

Example Code

FileSystemIterator

```
package com.headfirst.chapter9.iterator_compositePattern.iterator.filesyst

import java.io.File;

/**
 * Iterator for traversing files and directories in a file
        system.
 */
```

```
public class FileSystemIterator {
    private final File[] files;
    private int position = 0;

    public FileSystemIterator(File directory) {
        if (!directory.isDirectory()) {
            throw new IllegalArgumentException("Provided file is
        not a directory.");
        }
        this.files = directory.listFiles();
    }

    public boolean hasNext() {
    return position < files.length;
    }

    public File next() {
        if (!hasNext()) {
            throw new IllegalStateException("No more files.");
        }
        return files[position++];
    }
}
```

FileSystemTest

```
package com.headfirst.chapter9.iterator_compositePattern.iterator.filesyst

import java.io.File;

/**
 * Client to test FileSystemIterator.
 */
public class FileSystemTest {
    public static void main(String[] args) {
        File directory = new File("src/main/java/com/headfirst/
        chapter9/iterator_compositePattern/iterator/filesystem");

        if (!directory.exists() || !directory.isDirectory()) {
            System.out.println("Invalid directory path.");
            return;
        }

        FileSystemIterator iterator = new
        FileSystemIterator(directory);

        System.out.println("Files in the directory:");
        while (iterator.hasNext()) {
```

```
                File file = iterator.next();
                System.out.println("- " + file.getName());
            }
        }
}
```

More Examples

Example 1: Database Result Sets

Database APIs often use iterators to traverse query results, enabling uniform access to rows.

Example 2: UI Components

UI frameworks use iterators to traverse and manage collections of components like buttons and text fields.

Summary Table

Component	Responsibility
Iterator	Provides traversal methods (`hasNext()`, `next()`)
Concrete Iterator	Implements traversal logic for a specific collection
Aggregate	Defines a method to create an iterator
Concrete Aggregate	Implements the Aggregate and provides an iterator

---

The Iterator Pattern simplifies traversal while encapsulating collection structures, making it an essential tool for flexible and maintainable code.

Composite Pattern - Managing Hierarchies

What is the Composite Pattern?

The **Composite Pattern** is a structural design pattern that allows you to compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly. This is especially useful when dealing with recursive structures such as file systems, graphical UI components, or organizational hierarchies.

Key Components of the Composite Pattern

1. **Component Interface**:
    - Defines the common interface for all objects in the hierarchy.
2. **Leaf**:
    - Represents individual objects with no children.
3. **Composite**:
    - Represents objects that can hold other objects (both Leaf and Composite).
4. **Client**:
    - Interacts with objects through the Component interface.

Why Use the Composite Pattern?

- **Uniformity**: Treats individual and composite objects the same way.
- **Flexibility**: Easily adds new types of components or modifies the structure.
- **Scalability**: Handles complex hierarchical structures without modifying client code.

How It Works

1. Define a **Component Interface** with common methods for both Leafs and Composites.
2. Implement the **Leaf** class for objects that do not have children.
3. Implement the **Composite** class for objects that can contain other components.
4. Use the **Client** to interact with the Component interface.

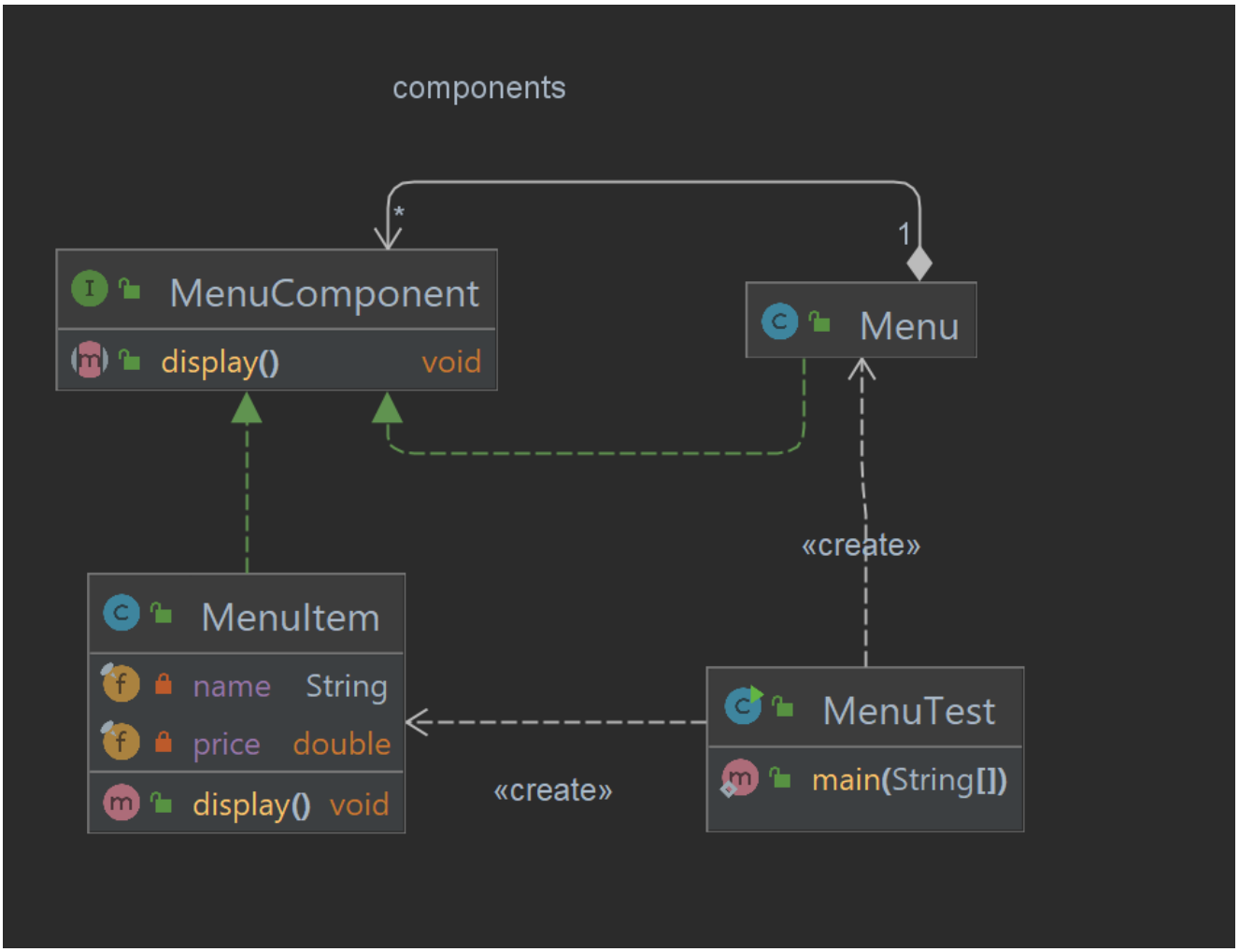
Project Structure

```
src/main/java/com/headfirst/chapter9/iterator_compositePattern/
composite
├── menu/
│   ├── MenuComponent.java          # Component Interface
│   ├── MenuItem.java               # Leaf
│   ├── Menu.java                   # Composite
│   └── MenuTest.java               # Client
└── filesystem/
    ├── FileComponent.java          # Component Interface
    ├── FileLeaf.java               # Leaf
    ├── DirectoryComposite.java     # Composite
    ├── FileSystemTest.java         # Client
```

Example: Menu Hierarchy

Consider a restaurant menu system where each menu can contain menu items (Leaf) or other menus (Composite). Clients can traverse the entire hierarchy uniformly.

Class Diagram



Example Code

Component Interface

```
public interface MenuComponent {
    void display();
}
```

Leaf

```
public class MenuItem implements MenuComponent {
    private final String name;
    private final double price;

    public MenuItem(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public void display() {
        System.out.println("Item: " + name + ", Price: $" +
        price);
    }
}
```

Composite

```
import java.util.ArrayList;
import java.util.List;

public class Menu implements MenuComponent {
    private final String name;
    private final List<MenuComponent> components = new
        ArrayList<>();

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

    public void add(MenuComponent component) {
        components.add(component);
    }

    public void remove(MenuComponent component) {
        components.remove(component);
    }
```

```
    @Override
    public void display() {
        System.out.println("Menu: " + name);
        for (MenuComponent component : components) {
            component.display();
        }
    }
}
```

Client

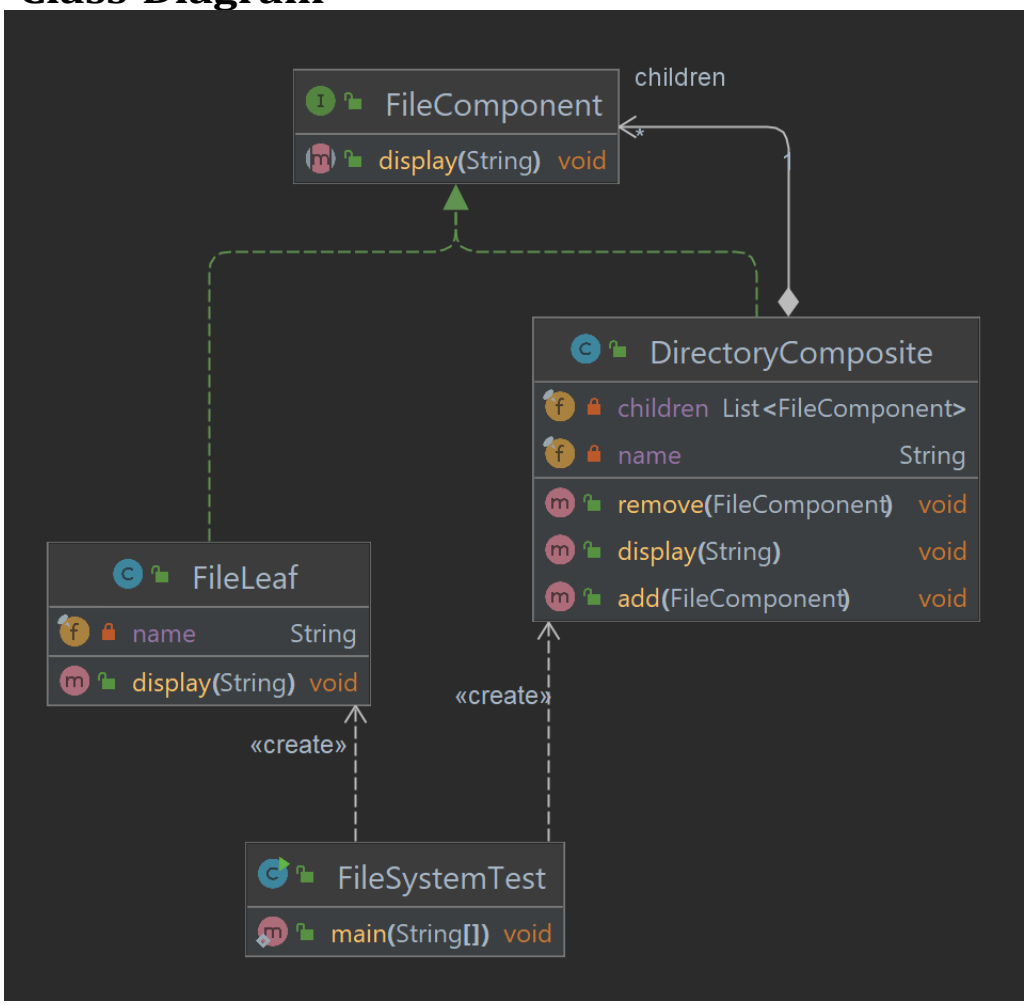
```
public class MenuTest {
    public static void main(String[] args) {
        // Create individual menu items
        MenuComponent pancake = new MenuItem("Pancake", 5.99);
        MenuComponent waffle = new MenuItem("Waffle", 6.99);
        MenuComponent coffee = new MenuItem("Coffee", 2.99);

        // Create breakfast menu and add items
        MenuComponent breakfastMenu = new Menu("Breakfast Menu");
        breakfastMenu.add(pancake);
        breakfastMenu.add(waffle);

        // Create main menu and add sub-menus and items
        MenuComponent mainMenu = new Menu("Main Menu");
        mainMenu.add(breakfastMenu);
        mainMenu.add(coffee);

        // Display the entire menu hierarchy
        mainMenu.display();
    }
}
```

Real-World Applications

Example: File Systems

File systems consist of folders (Composite) that can contain files (Leaf) or other folders. The Composite Pattern allows clients to treat files and folders uniformly.

Class Diagram



Example Code

File Component

```
package com.headfirst.chapter9.iterator_compositePattern.composite.filesys
```

File Leaf

```
package com.headfirst.chapter9.iterator_compositePattern.composite.filesys


/**
 * Leaf class representing a file.
 */
public class FileLeaf implements FileComponent {
    private final String name;

    public FileLeaf(String name) {
        this.name = name;
    }
```

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

Directory Composite

```
package com.headfirst.chapter9.iterator_compositePattern.composite.filesys

import java.util.ArrayList;
import java.util.List;

/**
 * Composite class representing a directory.
 */
public class DirectoryComposite implements FileComponent {
    private final String name;
    private final List<FileComponent> children = new
        ArrayList<>();

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

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

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

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

File System Test

```
package com.headfirst.chapter9.iterator_compositePattern.composite.filesys

/**
 * Client class to test the Composite Pattern for a file system.
```

```
 */
public class FileSystemTest {
    public static void main(String[] args) {
        // Create files
        FileComponent file1 = new FileLeaf("file1.txt");
        FileComponent file2 = new FileLeaf("file2.txt");
        FileComponent file3 = new FileLeaf("file3.txt");

        // Create directories and add files
        DirectoryComposite root = new DirectoryComposite("root");
        DirectoryComposite subDir1 = new
        DirectoryComposite("subDir1");
        DirectoryComposite subDir2 = new
         DirectoryComposite("subDir2");

        subDir1.add(file1);
        subDir1.add(file2);

        subDir2.add(file3);

        root.add(subDir1);
        root.add(subDir2);

        // Display the entire file system hierarchy
        root.display("");
    }
}
```

More Examples

Example 1: GUI Components

Graphical UI frameworks often have components like panels (Composite) that can contain other components (e.g., buttons, text fields) or panels.

Example 2: Organizational Hierarchies

Organizations have managers (Composite) with subordinates (Leaf or Composite), enabling uniform operations like calculating salaries or printing reports.

Summary Table

Component	Responsibility
Component Interface	Common interface for all objects
Leaf	Represents individual objects with no children
Composite	Holds and manages child components

The **Composite Pattern** is essential for managing hierarchical structures with consistent operations, enabling scalability and flexibility.