ustomer Requirement, Factory, and Bridge Patterns with Document Editor Exa

Scenario 2: Customer Request

Customer Request:

"I need a document editor that not only supports multiple document formats like Word, PDF, and Text but also allows me to apply different themes or styles to the documents (e.g., Dark Mode, Light Mode). I want to ensure that adding new themes in the future is easy, and I don?t want the document type to be tightly coupled with the theme or style."

Choosing the Right Pattern:

Pattern Recommendation: Bridge Pattern

Why?

The Bridge Pattern is suitable for this scenario because:

- Decoupling Abstraction and Implementation: The customer wants to separate the document type (Word, PDF, Text) from the theme or style (Dark Mode, Light Mode). The Bridge Pattern decouples these two dimensions, allowing them to vary independently.

- Flexibility in Extending Features: The Bridge Pattern allows you to add new document formats or themes without altering existing code. For example, you can introduce a new document type or a new theme without affecting the other.

- Simplifies Complex Systems: By decoupling document types and themes, you simplify the code structure, making it easier to manage and extend.

Step 1: Implementing the Factory Pattern

Initial Requirements:

- Create a document editor that can handle different document formats (e.g., Word, PDF).
- The operations include open(), save(), and close().

Factory Method Implementation:

```
// Interface for Documents
interface Doc {
    void open();
    void save();
    void close();
}

// Concrete Implementations
class WordDoc implements Doc {
    @Override
    public void open() {
        System.out.println("Opening Word document.");
    }
    @Override
    public void save() {
        System.out.println("Saving Word document.");
    }
    @Override
    public void close() {
        System.out.println("Closing Word document.");
    }
}

class PDFDoc implements Doc {
    @Override
    public void open() {
        System.out.println("Opening PDF document.");
    }
    @Override
    public void save() {
        System.out.println("Saving PDF document.");
    }
    @Override
    public void close() {
        System.out.println("Closing PDF document.");
    }
}

// Factory for Document Creation
abstract class DocCreator {
    abstract Doc createDoc();
}

class WordDocCreator extends DocCreator {
    @Override
    Doc createDoc() {
        return new WordDoc();
    }
}

class PDFDocCreator extends DocCreator {
    @Override
    Doc createDoc() {
        return new PDFDoc();
    }
}

// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
        DocCreator creator = new WordDocCreator(); // or new PDFDocCreator()
        Doc doc = creator.createDoc();
        doc.open();
        doc.save();
        doc.close();
    }
}
```

Explanation:

Factory Pattern is used here to encapsulate the creation of document objects. The client (DocumentEditor) doesn?t need to know about the specifics of WordDoc or PDFDoc; it just calls the createDoc() method.

Step 2: Extending to the Bridge Pattern

New Requirements:

- Add support for different themes (e.g., Dark Mode, Light Mode).
- The document types and themes should be decoupled so that you can easily add new themes or document types in the future.

Bridge Pattern Implementation:

```
// Interface for Themes
interface Theme {
    String color();
    String contrast();
}

// Concrete Implementations for Themes
class LightTheme implements Theme {
    @Override
    public String color() {
        return "Light Colors";
    }
    @Override
    public String contrast() {
        return "Low Contrast";
    }
}
```

```java
class DarkTheme implements Theme {
    @Override
    public String color() {
        return "Dark Colors";
    }
    @Override
    public String contrast() {
        return "High Contrast";
    }
}

// Updated Interface for Documents with Theme Support
abstract class Doc {
    protected Theme theme;

    public Doc(Theme theme) {
        this.theme = theme;
    }

    abstract void open();
    abstract void save();
    abstract void close();
}

// Concrete Implementations of Documents
class WordDoc extends Doc {
    public WordDoc(Theme theme) {
        super(theme);
    }

    @Override
    public void open() {
                System.out.println("Opening Word document with " + theme.color() + " and " +
theme.contrast());
    }
    @Override
    public void save() {
        System.out.println("Saving Word document with " + theme.color() + " and " + theme.contrast());
    }
    @Override
    public void close() {
        System.out.println("Closing Word document with " + theme.color() + " and " + theme.contrast());
    }
}

class PDFDoc extends Doc {
    public PDFDoc(Theme theme) {
        super(theme);
    }

    @Override
    public void open() {
                System.out.println("Opening PDF document with " + theme.color() + " and " +
theme.contrast());
    }
    @Override
    public void save() {
        System.out.println("Saving PDF document with " + theme.color() + " and " + theme.contrast());
    }
    @Override
    public void close() {
        System.out.println("Closing PDF document with " + theme.color() + " and " + theme.contrast());
    }
}

// Updated Factory with Theme Support
abstract class DocCreator {
    abstract Doc createDoc(Theme theme);
}

class WordDocCreator extends DocCreator {
    @Override
    Doc createDoc(Theme theme) {
        return new WordDoc(theme);
    }
}

class PDFDocCreator extends DocCreator {
    @Override
    Doc createDoc(Theme theme) {
        return new PDFDoc(theme);
    }
}

// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
        Theme darkTheme = new DarkTheme();
        DocCreator creator = new WordDocCreator();
        Doc doc = creator.createDoc(darkTheme);
        doc.open();
        doc.save();
        doc.close();
    }
}
```

Explanation:

Bridge Pattern is used to decouple the document type from the theme. The document class now depends on a theme, which allows different combinations of documents and themes to be used without modifying the existing code.
This pattern enables adding new themes or document types with minimal changes to the codebase.

Key Takeaways:

- Factory Pattern: Initially used to manage the creation of different document types, hiding the complexity of object creation from the client.
- Bridge Pattern: Applied when the requirement extended to include themes, allowing independent

variations in document types and themes.

This approach allows you to start with a simpler design (Factory Pattern) and evolve the design to handle more complex scenarios (Bridge Pattern) as new requirements emerge.