Lecture 8

Dr. Umair Rehman

Agenda

- Design Patterns Overview
- Singleton Pattern
  - Overview
  - Implementation
- Factory Pattern
- Abstract Factory Pattern

Design Patterns

- Reusable solutions to common problems that arise in software design.
- Not specific pieces of code but conceptual templates
- Help structure and solve design challenges.
- Promote code reusability, scalability, and maintainability by providing
  - Tested, proven development paradigms

Creational Patterns

- Object creation mechanisms.
- Singleton, Factory Method, Abstract Factory, Builder, Prototype.
- Decoupling object instantiation from implementation.

Structural Patterns

- Organizing relationships between objects.
- Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
- Efficient object composition and flexibility.

Behavioral Patterns

- Communication between objects.
- Observer, Strategy, Command, Chain of Responsibility, State, Template Method.
- Object interaction and responsibility delegation.

Architectural Patterns

- High-level organization of software systems.
- Layered, Microservices, Event-Driven, MVC, Client-Server, SOA.
- System structure and scalability

Singleton Pattern

- The Singleton pattern ensures that a class has only one instance
- Provides a global point of access to that instance.

Singleton Pattern

- Makes it impossible to create multiple instances.

Singleton Pattern Cons

- Global State: Hard to track changes.
- Tight Coupling: Harder to modify code.
- Difficult Testing: Hard to mock in tests.
- Hidden Dependencies: Dependencies are not obvious.
- Concurrency Issues: Risk of thread safety problems.

Singleton Pattern Pros

- Single Instance: Ensures only one object.
- Global Access: Central point for accessing the instance.
- Lazy Initialization: Creates instance only when needed.
- Controlled Resource Management: Prevents resource duplication.
- Consistency: Ensures uniform access to shared resources.

Step 1: Define a Class

```
class MySingleton {
    // Class contents will go here
};

```

Step 2: Make the constructor private

```cpp
class MySingleton {
private:
    MySingleton() {
        // Constructor is private
    }
};

```

Step 2: Make the constructor private

- Why private?
  - To prevent multiple instances
  - Enforce a single, globally accessible instance.

- What happens if not?
  - Multiple instances can be created

```
MySingleton obj = MySingleton();  // This will be blocked

```

Step 3: Create a static pointer to hold the single instance

```
class MySingleton {
private:
    static MySingleton* instance;  // Static pointer to hold the instance

    MySingleton() {
        // Private constructor
    }
};

```

Step 3: Create a static pointer to hold the single instance

- A static variable to store the single instance of the class.
- The "only" instance that gets created.
- Variable belongs to the class, not to any object,
  - Important because there is only one instance.

Step 4: Create a static method

```cpp
class MySingleton {
private:
    static MySingleton* instance;  // Static pointer to hold the instance

    MySingleton() {
        // Private constructor
    }

public:
    static MySingleton* getInstance() {
        if (instance == nullptr) {
            instance = new MySingleton();  // Create a new instance if it doesn't exist
        }
        return instance;
    }
};

```

Step 4: Create a static method

- We need a way to control how and when the instance is created.

- `getInstance()` is the only way to create or access the instance of the class

Step 5: Initialize the static instance to `nullptr`

```cpp
MySingleton* MySingleton::instance = nullptr;

```

- Initialize the static pointer to `nullptr` at the beginning
- `getInstance()` method can check whether the instance exists.

Step 6: Use the Singleton in the `main()` function

```cpp
int main() {
    MySingleton* s1 = MySingleton::getInstance();  // Creates the instance
    MySingleton* s2 = MySingleton::getInstance();  // Returns the same instance

    if (s1 == s2) {
        std::cout << "Both are the same instance!" << std::endl;
    }

    return 0;
}
```

- `s1` and `s2` both point to the same instance
- `getInstance()` ensures only one instance of `MySingleton` is ever created.

Complete Example

```cpp
#include <iostream>

class MySingleton {
private:
    static MySingleton* instance;  // Static pointer to hold the instance

    // Private constructor to prevent direct instantiation
    MySingleton() {
        std::cout << "Instance created!" << std::endl;
    }

public:
    // Static method to get the single instance
    static MySingleton* getInstance() {
        if (instance == nullptr) {
            instance = new MySingleton();  // Create the instance if it doesn't exist
        }
        return instance;
    }
};
```

Overview

- Private Constructor: Blocks external instantiation.
- Static Pointer: Stores and shares the single instance.
- Static getInstance(): Creates or returns the instance.
- Static Initialization: Initializes the instance as nullptr.

Why Object Creation is Important

- How and when objects are created has a significant impact
  - System's ability to evolve
  - Remain flexible
  - Maintain clean code.

Rigid Dependencies

- Directly constructing objects introduces rigid dependencies
  - Ties code to specific implementations
  - Makes extending the system later difficult

- Using patterns like the Factory Method allows acquisition of objects
  - No to worry how they are constructed, enabling flexibility

Tight Coupling

- Create objects using `new` directly inside your business logic
  - Code becomes tightly coupled to specific implementations

- Change the object type or add a new type later,
  - Introducing risk and reducing maintainability
  - Cannot easily swap out implementations

Tight Coupling

```cpp
CreditCardPayment* payment = new CreditCardPayment();
payment->process();

```

```cpp
PayPalPayment* payment = new PayPalPayment();
payment->process();

```

Why Hide Object Creation

- If object creation logic is mixed into business logic
  - Difficult to change later

- Hiding object creation in factories,
  - Isolate that responsibility and make the system more adaptable to changes

Example

- Imagine you are developing a document management system that can handle different types of documents, such as Word documents and PDF documents.
- You want to provide functionality for creating and printing these documents.
- Initially, you might be tempted to directly create instances of these document classes in the client code.

Example

```cpp
#include <iostream>
#include <string>

// A simple document class
class WordDocument {
public:
    void print() {
        std::cout << "Printing Word document..." << std::endl;
    }
};

class PDFDocument {
public:
    void print() {
        std::cout << "Printing PDF document..." << std::endl;
    }
};
```

Example

```cpp
int main() {
    std::string docType;
    std::cout << "Enter document type (Word/PDF): ";
    std::cin >> docType;

    if (docType == "Word") {
        WordDocument* doc = new WordDocument();
        doc->print();
        delete doc;
    } else if (docType == "PDF") {
        PDFDocument* doc = new PDFDocument();
        doc->print();
        delete doc;
    } else {
        std::cout << "Unknown document type!" << std::endl;
    }

    return 0;
}
```

Tight Coupling

- `main()` directly creates WordDocument or PDFDocument.

- Introduce a new document type, such as SpreadsheetDocument
  - Modify this function
  - Tightly couples the business logic to specific document types

Lack of Flexibility

- You want to support a new document type
  - Update all the places where object creation occurs
  - With growth hard to manage.

- Factory Method allows us to hide the object creation logic behind a factory
  - System more flexible and easier to extend

Exercise

- Real-world scenarios where object creation is an issue (e.g., creating objects based on user input, such as different types of products in an e-commerce system).

Factory Method

- Provides a way to delegate the creation of objects to subclasses
- Client code doesn’t depend on the specifics of object creation

Factory Method

- Defines an interface for creating an object
- But allows subclasses to decide which class to instantiate

Factory Method

- Creation of the object is abstracted from the client
- Client only interacts with abstract types
- Not concrete implementations

Example

- Different types of documents like WordDocument and PDFDocument.

- Using a DocumentFactory
    - The factory decides which document to create based on input from the user

Separation of Concerns

- Object creation is separated from the core logic
- Principle of Separation of Concerns
- Client no longer needs to know how objects are created
- Or which specific class to instantiate

Single Responsibility Principle (SRP)

- Client code is only responsible for using the object
- Not for deciding how it should be created.
- Factory takes care of object creation
- Better organization and cleaner code

Polymorphism in Action

- Client code interacts with abstract types (e.g., Document)

- Rather than specific concrete implementations (e.g., WordDocument or PDFDocument).

Polymorphism in Action

- Greater flexibility
- Client code can work with any new type of document
- Without being modified—thanks to polymorphism

Polymorphism in Action

- Client code works with a Document* object
- Calls methods like print() or save()
- Not knowing whether the object is a WordDocument or PDFDocument
- Specific object type is hidden behind the factory

Constructor Overloading vs. Factory

- You might think of using constructor overloading
- Manage different ways of creating objects.
- You could overload a constructor to take a type parameter ("Word", "PDF") and instantiate different objects

Constructor Overloading vs. Factory

- Violates the Single Responsibility Principle (SRP)

- Class is now responsible for both
  - Object creation
  - Its core behavior.

- Mixes the logic of object creation into the class
  - Code less flexible and harder to maintain

Constructor Overloading vs. Factory

- We work with different types, like Word or PDF.
- Factories create the right type of document for us.
- The document knows how to "print" itself, no matter the type.

Factory Method

- Imagine you are developing a document management system that can handle different types of documents, such as Word documents and PDF documents.
- You want to provide functionality for creating and printing these documents.
- Initially, you might be tempted to directly create instances of these document classes in the client code.

Code Walkthrough

```cpp
// Step 1: Define the abstract base class for Document
class Document {
public:
    virtual void print() = 0;  // Pure virtual function for printing
    virtual ~Document() {}     // Virtual destructor for cleanup
};

```

Code Walkthrough

```cpp
// Step 2: Concrete implementations of WordDocument and PDFDocument
class WordDocument : public Document {
public:
    void print() override {
        std::cout << "Printing Word document..." << std::endl;
    }
};

class PDFDocument : public Document {
public:
    void print() override {
        std::cout << "Printing PDF document..." << std::endl;
    }
};
```

Code Walkthrough

```cpp
// Step 3: Define the abstract DocumentFactory class
class DocumentFactory {
public:
    virtual Document* createDocument() = 0;  // Factory method to create a document
    virtual ~DocumentFactory() {}
};

```

Code Walkthrough

```cpp
// Step 4: Concrete factories for WordDocument and PDFDocument
class WordDocumentFactory : public DocumentFactory {
public:
    Document* createDocument() override {
        return new WordDocument();  // Creates a WordDocument
    }
};

class PDFDocumentFactory : public DocumentFactory {
public:
    Document* createDocument() override {
        return new PDFDocument();  // Creates a PDFDocument
    }
};

```

Code Walkthrough

```cpp
// Step 5: Main function demonstrating Factory Method
int main() {
    std::string docType;
    std::cout << "Enter document type (Word/PDF): ";
    std::cin >> docType;

    DocumentFactory* factory = nullptr;

    // Decide which factory to use based on user input
    if (docType == "Word") {
        factory = new WordDocumentFactory();
    } else if (docType == "PDF") {
        factory = new PDFDocumentFactory();
    } else {
        std::cout << "Unknown document type!" << std::endl;
        return 1;
    }

    // Use the factory to create the document
    Document* document = factory->createDocument();
    document->print();  // Call the print method

    // Clean up
    delete document;
    delete factory;

    return 0;
```

Explanation

- The user inputs the document type (Word or PDF).
- Depending on the input, the appropriate factory (WordDocumentFactory or PDFDocumentFactory) is created.
- The factory then creates the document, and its print() method is called.
- After usage, both the document and factory are cleaned up with delete to avoid memory leaks.

Explanation

- Document is an abstract base class that defines the print() method.
- WordDocument and PDFDocument are concrete implementations
  - Override the print() method
- The DocumentFactory defines a method for creating Document objects
  - Leaves the actual implementation to its subclasses (WordDocumentFactory and PDFDocumentFactory)
- Client code interacts with a Document* pointer
  - Doesn't know if it's a WordDocument or a PDFDocument
  - This is handled by the factory.

Exercise

- Implement a Factory Method example where they create shapes like Circle and Square.

- Implement a ShapeFactory that returns the appropriate shape based on user input.

Exercise

- Steps:
  - Create an abstract base class Shape with a method draw().
  - Implement concrete classes Circle and Square that override draw().
  - Implement a ShapeFactory class with a method createShape() that returns a Shape object.
  - Main function, ask the user for the shape type (Circle/Square) and use the factory to create and draw the shape.

Benefits

- Supports extensibility
- Add new types of objects (e.g., a new document type or vehicle type)
    - Creating a new factory (SpreadsheetDocumentFactory)
- Without modifying the existing client code or the factory’s core logic.

Benefits

- This adheres to the Open/Closed Principle,
- Factory Method pattern allows new classes to be added (extension)
- Without needing to modify the client code (closure)

When It's Overkill

- Few Object Types that rarely change
  - Factory Method may add unnecessary complexity
  - Manually creating these objects using new might be simpler
  - More efficient than adding a factory layer

- When object creation is simple
  - Doesn't require delegation or abstraction
  - The Factory Method might add more layers than needed

Extension

- You are building a document management system for a company that handles multiple types of documents. The company primarily works with Word documents, PDF documents, and Spreadsheets. Each document type has its own specific functionalities, such as printing. The system needs to be flexible enough to handle future extensions, such as adding new document types.

Code Walkthrough – Extension

```cpp
// Step 1: Define the abstract base class for Document
class Document {
public:
    virtual void print() = 0;  // Pure virtual function for printing
    virtual ~Document() {}     // Virtual destructor for cleanup
};

```

Code Walkthrough – Extension

```
// Step 2: Concrete implementations of WordDocument, PDFDocument, and SpreadsheetDocument
class WordDocument : public Document {
public:
    void print() override {
        std::cout << "Printing Word document..." << std::endl;
    }
};

class PDFDocument : public Document {
public:
    void print() override {
        std::cout << "Printing PDF document..." << std::endl;
    }
};

class SpreadsheetDocument : public Document {
public:
    void print() override {
        std::cout << "Printing Spreadsheet document..." << std::endl;
    }
};
```

Code Walkthrough – Extension

```cpp
// Step 3: Define the abstract DocumentFactory class
class DocumentFactory {
public:
    virtual Document* createDocument() = 0;  // Factory method to create a document
    virtual ~DocumentFactory() {}
};

```

Code Walkthrough – Extension

```cpp
// Step 4: Concrete factories for WordDocument, PDFDocument, and SpreadsheetDocument
class WordDocumentFactory : public DocumentFactory {
public:
    Document* createDocument() override {
        return new WordDocument();  // Creates a WordDocument
    }
};

class PDFDocumentFactory : public DocumentFactory {
public:
    Document* createDocument() override {
        return new PDFDocument();  // Creates a PDFDocument
    }
};

class SpreadsheetDocumentFactory : public DocumentFactory {
public:
    Document* createDocument() override {
        return new SpreadsheetDocument();  // Creates a SpreadsheetDocument
    }
};
```

Code Walkthrough – Extension

```cpp
// Step 5: Main function demonstrating Factory Method with a new document type
int main() {
    std::string docType;
    std::cout << "Enter document type (Word/PDF/Spreadsheet): ";
    std::cin >> docType;

    DocumentFactory* factory = nullptr;

    // Decide which factory to use based on user input
    if (docType == "Word") {
        factory = new WordDocumentFactory();
    } else if (docType == "PDF") {
        factory = new PDFDocumentFactory();
    } else if (docType == "Spreadsheet") {
        factory = new SpreadsheetDocumentFactory();
    } else {
        std::cout << "Unknown document type!" << std::endl;
        return 1;
    }

    // Use the factory to create the document
    Document* document = factory->createDocument();
    document->print();  // Call the print method

```

Explanation

- Create a factory that can return different types of vehicles, such as Car, Bike, and Truck.

- Create an abstract base class Vehicle with a method drive().
  - Implement concrete classes Car, Bike, and Truck that override drive().
  - Implement a VehicleFactory class that decides which vehicle to create based on user input.
  - Ensure the system is extensible so we can add new vehicle types later
  - Without modifying the client code

Abstract Factory Method

- Multiple related objects need to be created together

Abstract Factory Method

- Set of related products (e.g., documents and editors) that belong to the same family

- Hiding the details of how these objects are created

Abstract Factory Method

- Imagine a document creation system where a
  - WordDocument needs to be edited using a WordEditor
  - PDFDocument requires a PDFEditor.
- Abstract Factory pattern helps create these related objects (document + editor)
- in a consistent way without needing to know exact implementation

Why Use Abstract Factory

- Multiple related objects that must work together.
- Products from one family (e.g., WordDocument and WordEditor) are created together
- Prevents products from different families (e.g., WordEditor and PDFDocument) from being mixed up

When It’s Useful

- Families of objects evolve and more types of products are added over time

Factory Method vs. Abstract Factory

- Factory Method allows you to create one type of object at a time (e.g., WordDocument

- Abstract Factory is used to create multiple related objects (e.g., WordDocument and WordEditor) that must work together

Factory Method vs. Abstract Factory

- Factory Method
  - A DocumentFactory creates a WordDocument or a PDFDocument

- Abstract Factory creates both a
  - WordDocument and a WordEditor
  - PDFDocument and a PDFEditor

Abstract Factory

- You are building a document management system that not only creates documents (such as Word and PDF documents) but also provides corresponding editors for each document type.
- The WordDocument requires a WordEditor, and the PDFDocument requires a PDFEditor.
- You want to ensure that these related objects (documents and editors) are created together to avoid mismatches (e.g., creating a WordDocument and a PDFEditor).
- The system should be flexible enough to allow for the future addition of new document-editor pairs (such as SpreadsheetDocument with a SpreadsheetEditor) without requiring changes to the core system.

Explanation

```cpp
// Step 1: Define abstract base classes for Document and Editor
class Document {
public:
    virtual void print() = 0;  // Pure virtual function for printing
    virtual ~Document() {}     // Virtual destructor for cleanup
};

class Editor {
public:
    virtual void open() = 0;   // Pure virtual function for opening the editor
    virtual ~Editor() {}       // Virtual destructor for cleanup
};
```

Explanation

```cpp
// Step 2: Concrete implementations of WordDocument and WordEditor
class WordDocument : public Document {
public:
    void print() override {
        std::cout << "Printing Word document..." << std::endl;
    }
};

class WordEditor : public Editor {
public:
    void open() override {
        std::cout << "Opening Word editor..." << std::endl;
    }
};

```

Explanation

```cpp
// Step 3: Concrete implementations of PDFDocument and PDFEditor
class PDFDocument : public Document {
public:
    void print() override {
        std::cout << "Printing PDF document..." << std::endl;
    }
};

class PDFEditor : public Editor {
public:
    void open() override {
        std::cout << "Opening PDF editor..." << std::endl;
    }
};
```

Explanation

```cpp
// Step 4: Define the abstract factory interface
class DocumentEditorFactory {
public:
    virtual Document* createDocument() = 0;  // Create a document
    virtual Editor* createEditor() = 0;      // Create an editor
    virtual ~DocumentEditorFactory() {}
};

```

Explanation

```
// Step 5: Concrete factories for Word and PDF families
class WordDocumentEditorFactory : public DocumentEditorFactory {
public:
    Document* createDocument() override {
        return new WordDocument();  // Creates a WordDocument
    }

    Editor* createEditor() override {
        return new WordEditor();    // Creates a WordEditor
    }
};

class PDFDocumentEditorFactory : public DocumentEditorFactory {
public:
    Document* createDocument() override {
        return new PDFDocument();   // Creates a PDFDocument
    }

    Editor* createEditor() override {
        return new PDFEditor();     // Creates a PDFEditor
    }
```

Explanation

```cpp
// Step 6: Main function demonstrating Abstract Factory
int main() {
    std::string docType;
    std::cout << "Enter document type (Word/PDF): ";
    std::cin >> docType;

    DocumentEditorFactory* factory = nullptr;

    // Decide which factory to use based on user input
    if (docType == "Word") {
        factory = new WordDocumentEditorFactory();
    } else if (docType == "PDF") {
        factory = new PDFDocumentEditorFactory();
    } else {
        std::cout << "Unknown document type!" << std::endl;
        return 1;
    }

    // Use the factory to create both document and editor
    Document* document = factory->createDocument();
    Editor* editor = factory->createEditor();

    // Work with the document and editor
    editor->open();
    document->print();
```

Code Explanation

- Abstract Factory
  - The DocumentEditorFactory defines an interface for creating families of objects (document + editor).
- Concrete Factories
  - WordDocumentEditorFactory and PDFDocumentEditorFactory create related objects (WordDocument and WordEditor, or PDFDocument and PDFEditor).
- Client code interacts with Document* and Editor*
  - Without knowing the concrete types of the objects
  - Allowing flexibility and scalability

Extensibility with Abstract Factory

- Add new families of products (e.g., SpreadsheetDocument and SpreadsheetEditor) without modifying existing factories or client code.

- The client code that asks for a document and editor would not need to be changed, ensuring adherence to the Open/Closed Principle.

Abstract Factory Pitfalls

- Abstract Factory is overkill for unrelated object types.
- Use simpler designs if you have few product families and no plans to add more.
- Avoid Over-Engineering: Don't add complexity with Abstract Factory if multiple related objects aren't needed.

Activity

- Implement Abstract Factory example
  - create families of related products such as phones and chargers.
- Steps:
  - Define abstract base classes for Phone and Charger with methods like makeCall() and charge().
  - Create concrete implementations like AndroidPhone, iPhone, AndroidCharger, and iPhoneCharger.
  - Implement an AbstractPhoneChargerFactory that can create both a Phone and its corresponding Charger.
  - Add new phone types and chargers later to demonstrate extensibility.

Conclusion

- The Abstract Factory pattern is used to
  - Create families of related objects (e.g., documents and editors)
  - Without needing to specify their exact types.

- Abstract Factory is useful when creating multiple related products,

- Can be overkill for systems with few product families or unrelated objects.

Conclusion

- Creational Design Patterns
  - Singleton
  - Factory Method
  - Abstract Factory Method