Contents

What are Design Patterns?

Design patterns provide reusable solutions to recurring software design problems. Instead of solving problems from scratch, developers can apply proven templates to save time and effort.

Key Characteristics of Design Patterns:

1. **Reusable**: Solutions are general and can be reused across projects.
2. **Scalable**: Patterns improve scalability, making systems more modular.
3. **Readable**: Patterns improve code organization and maintainability.

Types of Design Patterns

Design patterns are divided into three categories:

1. **Creational**: Patterns for object creation.
2. **Structural**: Patterns for organizing classes and objects.
3. **Behavioral**: Patterns for managing communication between objects.

---

1. Creational Patterns

Singleton Pattern

Intent:

Ensure that a class has **only one instance** and provide a **global access point** to that instance.

Example Scenarios:

1. **Database Connection**:
   - A system requires only one database connection to optimize resource use.
2. **Logger Class**:
   - A single logger instance writes logs to a file or console.

C++ Implementation:

Here is an example of the Singleton pattern in C++:

```cpp
#include <iostream>
using namespace std;

class Singleton {
private:
    static Singleton* instance; // Static instance of the class
    Singleton() {}              // Private constructor

public:
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }

    void showMessage() {
        cout << "Singleton Instance: Only one instance exists!" << endl;
    }
};

// Initialize static member
Singleton* Singleton::instance = nullptr;

// Main Function
int main() {
    Singleton* s1 = Singleton::getInstance();
    Singleton* s2 = Singleton::getInstance();

    s1->showMessage();

    // Checking if both instances are the same
    if (s1 == s2) {
        cout << "Both pointers point to the same instance." << endl;
    }
    return 0;
}
```

Output:

```
Singleton Instance: Only one instance exists!
Both pointers point to the same instance.
```

Real-World Application: Singleton in Logging Systems

- **Problem**: Logs need to be stored centrally across multiple components of an application.
- **Solution**: Use a Singleton pattern to create a **single logger instance**.

System Example:

- A **banking system** that logs transactions uses the Singleton pattern for a logging module.

---

2. Structural Patterns

Adapter Pattern

Intent:

Convert the interface of one class into an interface that clients expect. This pattern allows incompatible interfaces to work together.

Example Scenario:

Problem:
You have an old legacy system that computes **investment returns**. You want to integrate it into a modern application, but the interfaces are incompatible.

Solution:
The **Adapter Pattern** acts as a bridge to integrate incompatible systems.

C++ Implementation:

```cpp
#include <iostream>
using namespace std;

// Old System (Adaptee)
class LegacySystem {
public:
    void calculateInvestment() {
        cout << "Calculating investment using legacy system..." << endl;
    }
};
```

```
// New Interface (Target)
class NewSystemInterface {
public:
    virtual void performInvestmentCalculation() = 0;
};

// Adapter Class
class Adapter : public NewSystemInterface {
private:
    LegacySystem* legacy;

public:
    Adapter(LegacySystem* l) : legacy(l) {}

    void performInvestmentCalculation() override {
        legacy->calculateInvestment(); // Adapting the legacy function
    }
};

// Main Function
int main() {
    LegacySystem* legacySystem = new LegacySystem();
    NewSystemInterface* adapter = new Adapter(legacySystem);

    adapter->performInvestmentCalculation(); // Modern system calls through the
adapter

    delete legacySystem;
    delete adapter;
    return 0;
}
```

Output:

```
Calculating investment using legacy system...
```

Real-World Example: Power Adapter

- A European power adapter converts a **220V socket** (legacy) to a **110V interface** for U.S. appliances.

Software Application:

- Integrating a **payment gateway** with an existing **e-commerce system** using an adapter to align interfaces.

---

3. Behavioral Patterns

Observer Pattern

Intent:

Define a one-to-many dependency between objects so that when one object (subject) changes state, all its dependents (observers) are notified automatically.

Example Scenario:

Problem:
A weather station provides real-time weather updates. Multiple devices (current conditions display, forecast display) need to stay updated whenever the weather changes.

Solution:
The **Observer Pattern** ensures all displays (observers) are notified automatically.

C++ Implementation:

```
#include <iostream>
#include <vector>
using namespace std;

// Observer Interface
class Observer {
public:
    virtual void update(float temperature) = 0;
};

// Subject Class
class WeatherStation {
private:
    vector<Observer*> observers;
    float temperature;
```

```
public:
    void addObserver(Observer* o) {
        observers.push_back(o);
    }

    void removeObserver(Observer* o) {
        observers.erase(remove(observers.begin(), observers.end(), o),
observers.end());
    }

    void setTemperature(float temp) {
        temperature = temp;
        notifyObservers();
    }

    void notifyObservers() {
        for (Observer* o : observers) {
            o->update(temperature);
        }
    }
};

// Concrete Observer 1
class DisplayDevice : public Observer {
public:
    void update(float temperature) override {
        cout << "Display Updated: Temperature is " << temperature << "°C" <<
endl;
    }
};

// Main Function
int main() {
    WeatherStation station;

    DisplayDevice device1, device2;

    station.addObserver(&device1);
    station.addObserver(&device2);

    station.setTemperature(25.5);

    station.removeObserver(&device2);
    station.setTemperature(30.0);
```

```
    return 0;
}
```

Output:

```
Display Updated: Temperature is 25.5°C
Display Updated: Temperature is 25.5°C
Display Updated: Temperature is 30°C
```

Real-World Example: Weather Systems

- A **weather application** sends updates to mobile devices, web dashboards, and notification systems whenever weather data changes.

Software Applications of Patterns

Pattern	Application
Singleton	Database connections, loggers, configuration files.
Adapter	Integrating legacy systems, payment gateways.
Observer	Event listeners, real-time dashboards, UI updates.

Summary

- **Design Patterns** offer ready-to-use templates to solve common design problems.
- **Singleton**: Ensures only one instance exists; useful for **resource control**.
- **Adapter**: Bridges incompatible interfaces; useful for **system integration**.
- **Observer**: Ensures automatic updates to dependents; useful for **event-driven systems**.

Detailed Notes for Lecture 19: GRASP Patterns

Lecture Outline

The lecture introduces the **GRASP Patterns** (General Responsibility Assignment Software Patterns), which are essential for Object-Oriented Design (OOD). It covers:

1. **Creator**
2. **Information Expert**
3. **Low Coupling**
4. **High Cohesion**
5. **Polymorphism**
6. **Pure Fabrication**

Responsibility-Driven Design (RDD)

In RDD, **objects** have two types of responsibilities:

1. **Doing**:
   - Performing actions such as calculations, creating objects.
   - Coordinating actions in other objects.
2. **Knowing**:
   - Managing encapsulated data.
   - Knowing related objects or derived information.

Abstraction:
Responsibilities are implemented via **methods** in classes.

What are GRASP Patterns?

GRASP patterns provide principles for assigning responsibilities in OOD systems.

Core GRASP Patterns:

1. **Creator**: Who should create objects?
2. **Information Expert**: Which class has the information to fulfill responsibilities?
3. **Low Coupling**: Minimize dependencies between objects.
4. **High Cohesion**: Maintain focus on a single task in a class.
5. **Polymorphism**: Behavior depends on object type.

6. **Pure Fabrication**: Create utility classes when domain-based solutions violate cohesion or coupling.

---

1. Creator Pattern

Intent:

- Assign responsibility for creating an object to the **class best suited** to do so.

Principle:

Class B should create class A if:

1. B **contains** or **aggregates** instances of A
2. B **records** instances of A
3. B **closely collaborates** with A

Monopoly Example:

Who creates **Square** objects?

- **Solution**: The **Board** creates Square objects because:
    - The **Board aggregates** all squares.
    - The Board **has information** about squares.

Static Model:

```
class Square {
public:
    Square(string name) : name(name) {}
    string name;
};

class Board {
private:
    vector<Square> squares;

public:
    void createSquares() {
        for (int i = 0; i < 40; i++) {
            squares.push_back(Square("Square" + to_string(i + 1)));
        }
    }
};
```

---

2. Information Expert Pattern

Intent:

Assign responsibilities to the class that has the **information** necessary to fulfill them.

Example:

A **Player's Marker** needs to locate the square it will move to.

- The **Board** is responsible for providing the square because it:
  - **Aggregates** all square information.
  - Has the required knowledge.

Code Example:

```
class Board {
private:
    map<string, Square> squares;

public:
    Square getSquare(string name) {
        return squares[name];
    }
};

class Player {
    void moveMarker(Board board, string squareName) {
        Square target = board.getSquare(squareName);
        cout << "Moving to: " << target.name << endl;
    }
};
```

---

3. Low Coupling

Definition:

Coupling measures how **interdependent** objects are.

Goal:

Assign responsibilities to minimize **unnecessary dependencies**.

Why is Low Coupling Important?

- Reduces the **impact of changes** in one class on others.
- Increases **reusability** and maintainability.

Example:
In a **Payment System**, coupling is minimized by assigning responsibilities to classes logically tied to one another.

```
class Sale {
public:
    void addPayment(float amount) {
        Payment payment(amount);
    }
};

class Payment {
public:
    Payment(float amount) {
        cout << "Payment of " << amount << " added." << endl;
    }
};
```

- The **Sale** class is naturally coupled to **Payment**, as it logically owns payments.

---

4. High Cohesion

Definition:

Cohesion measures how well a class's responsibilities align with a **single purpose**.

Problem:

Low cohesion occurs when a class:

- Manages unrelated responsibilities.
- Performs too many tasks.

Solution: Assign responsibilities so that classes focus on **specific tasks**.

Example:

In the **MonopolyGame** class:

- Poor cohesion: One class handles the entire game flow.

```
class MonopolyGame {
    void playGame() {
        doA();
        doB();
        doC();
    }
};
```

- Improved cohesion: Break into smaller, focused classes.

```
class TurnManager {
    void takeTurn() {
        cout << "Player taking turn." << endl;
    }
};

class GameController {
    void playGame() {
        TurnManager turn;
        turn.takeTurn();
    }
};
```

---

5. Polymorphism

Definition:

Polymorphism allows objects to respond **differently** based on their type.

Problem:

Using **if-else** or **switch** statements to determine behavior violates OOD principles.

Solution: Use polymorphic method calls to handle variations.

Example:

In a Monopoly game, different squares have different actions.

```
class Square {
public:
    virtual void landedOn() = 0;
};

class PropertySquare : public Square {
public:
    void landedOn() override {
        cout << "Property: Pay rent or buy!" << endl;
    }
};

class GoSquare : public Square {
public:
    void landedOn() override {
        cout << "Go: Collect $200!" << endl;
    }
};
```

---

6. Pure Fabrication

Intent:

Assign responsibilities to an **artificial class** when solutions violate:

- High cohesion
- Low coupling

Problem:

Rolling dice in a game **doesn't logically belong** to the Player or Board classes.

Solution: Introduce a **Pure Fabrication** class to handle dice rolling.

```
class Dice {
public:
    int roll() {
        return rand() % 6 + 1;
    }
};
```

Benefits of Pure Fabrication:

1. **High Cohesion**: Responsibility is cleanly factored into a new class.
2. **Reusability**: The **Dice** class can be reused in other games.

Key Takeaways

1. **GRASP Patterns** provide principles for assigning responsibilities:
    - **Creator**: Who creates objects?
    - **Information Expert**: Assign tasks to classes with relevant information.
    - **Low Coupling**: Reduce dependencies.
    - **High Cohesion**: Keep classes focused.
    - **Polymorphism**: Use type-based behavior.
    - **Pure Fabrication**: Create utility classes for isolated tasks.
2. Applying GRASP patterns improves:
    - **Maintainability**
    - **Scalability**
    - **Reusability**

Case Study: Library Management System

We will now use the **GRASP Patterns** to design a **Library Management System** (LMS). This example will help you understand how each pattern applies to a real-world scenario.

Library Management System Overview

Problem Statement:

A Library Management System manages:

1. **Books** in the library.
2. **Patrons** (library members).
3. **Borrowing and Returning Books**.
4. **Fines** for overdue books.

The system should:

- Track books and their availability.
- Allow patrons to borrow or return books.
- Calculate overdue fines.
- Ensure the system is modular and easy to maintain.

Step-by-Step Application of GRASP Patterns

1. Creator Pattern

Problem: Who should create **Book** and **BorrowTransaction** objects?

Solution:

- The **Library** class creates **Book** objects because it **aggregates** all books.
- The **Library** class also creates **BorrowTransaction** objects because it keeps track of borrowing records.

Example Implementation:

```
class Book {
public:
    string title, author;
    bool isAvailable;

    Book(string t, string a) : title(t), author(a), isAvailable(true) {}
};

class BorrowTransaction {
public:
    string patronName;
    Book* borrowedBook;

    BorrowTransaction(string patron, Book* book) {
        patronName = patron;
        borrowedBook = book;
        borrowedBook->isAvailable = false;
    }
};

class Library {
    vector<Book> books;

public:
    void addBook(string title, string author) {
        books.push_back(Book(title, author));
    }

    BorrowTransaction* createBorrowTransaction(string patronName, string
bookTitle) {
        for (auto& book : books) {
```

```
            if (book.title == bookTitle && book.isAvailable) {
                return new BorrowTransaction(patronName, &book);
            }
        }
        return nullptr; // No available book found
    }
};
```

Explanation:

- The **Library** class uses the **Creator Pattern** to create and manage books and borrowing transactions.

---

2. Information Expert Pattern

Problem: Who knows whether a book is available for borrowing?

Solution:

- The **Book** class is the **Information Expert** because it knows about its availability.

Example Implementation:

```
class Book {
public:
    string title, author;
    bool isAvailable;

    Book(string t, string a) : title(t), author(a), isAvailable(true) {}

    bool checkAvailability() {
        return isAvailable;
    }
};
```

Explanation:

- The **Book** class has the information to answer the availability question.

---

3. Low Coupling

Problem: How to minimize dependencies between classes?

Solution:

- Assign responsibilities so that classes depend only on essential information.
- Avoid unnecessary coupling.

Example:

The **Library** class interacts with the **Book** class and **BorrowTransaction**, but it does not need to know the internal details of the book.

```
class Library {
public:
    vector<Book> books;

    void displayAvailableBooks() {
        for (const auto& book : books) {
            if (book.checkAvailability()) {
                cout << "Available: " << book.title << " by " << book.author
<< endl;
            }
        }
    }
};
```

Result:

- The Library depends on the **public methods** of the Book class.
- If the internal implementation of Book changes, the Library is unaffected.

---

4. High Cohesion

Problem: How to keep classes focused on single responsibilities?

Solution:

- Assign a single responsibility to each class.

Example:

1. **Book Class**: Manages book information.
2. **Patron Class**: Manages member information.
3. **Library Class**: Coordinates borrowing and returning books.
4. **FineCalculator Class**: Calculates overdue fines.

```
class FineCalculator {
public:
    static double calculateFine(int overdueDays) {
        return overdueDays * 0.5; // $0.5 per day
    }
};
```

---

5. Polymorphism

Problem: Different types of books require different actions when borrowed.

Solution:

- Use polymorphism to manage book-specific behaviors.

Example:

```
class Book {
public:
    virtual void borrowBook() = 0; // Pure virtual function
};

class RegularBook : public Book {
public:
    void borrowBook() override {
        cout << "Borrowing a regular book." << endl;
    }
};

class RareBook : public Book {
public:
    void borrowBook() override {
        cout << "Borrowing a rare book. Handle with care!" << endl;
    }
};
```

Result:

- When borrowing a book, the correct action is selected based on the book type.

---

6. Pure Fabrication

Problem: Who should handle fine calculation? This is not directly tied to any single class.

Solution:

- Introduce a **FineCalculator** class as a **Pure Fabrication** to calculate fines.

```
class FineCalculator {
public:
    static double calculateFine(int overdueDays) {
        return overdueDays * 1.0; // $1 per day
    }
};
```

Final System Design Summary

Class	Responsibility	Pattern Applied
Library	Manages books and borrowing transactions	Creator
Book	Manages book information and availability	Information Expert
BorrowTransaction	Represents borrowing details (patron, book borrowed)	Creator
Patron	Manages member details	High Cohesion
FineCalculator	Calculates fines for overdue books	Pure Fabrication
Book Subclasses	Handles book-specific behaviors (e.g., RareBook)	Polymorphism

---

Code Integration Example

```
int main() {
    Library library;

    // Add books
    library.addBook("1984", "George Orwell");
    library.addBook("To Kill a Mockingbird", "Harper Lee");

    // Borrow a book
    BorrowTransaction* transaction = library.createBorrowTransaction("John
Doe", "1984");
    if (transaction) {
        cout << "Book borrowed successfully by " << transaction->patronName
<< endl;
    } else {
        cout << "Book not available." << endl;
    }

    // Display available books
    library.displayAvailableBooks();

    // Fine calculation
    cout << "Fine for 5 overdue days: $" << FineCalculator::calculateFine(5)
<< endl;

    return 0;
}
```

Output:

```
Book borrowed successfully by John Doe
Available: To Kill a Mockingbird by Harper Lee
Fine for 5 overdue days: $2.5
```

Summary of GRASP in the Library System

1. **Creator**:
    - Library creates Books and BorrowTransactions.
2. **Information Expert**:
    - Book knows its availability.
3. **Low Coupling**:
    - Library interacts with Book through minimal public interfaces.
4. **High Cohesion**:
    - FineCalculator focuses only on fine calculation.
5. **Polymorphism**:
    - Book subclasses handle specific borrow actions.
6. **Pure Fabrication**:
    - FineCalculator is introduced to keep responsibilities focused.

Benefits of Applying GRASP

- **Maintainability**: Each class has a single, well-defined responsibility.
- **Scalability**: Adding new features (e.g., new book types) is easier.
- **Reusability**: FineCalculator and Book classes can be reused in other systems.