Lecture 11

Dr. Umair Rehman

Agenda

- Structural Design Patterns Overview
- Adapter Design Pattern
- Benefits and Limitations
- Adapter Design Pattern vs
  - Multiple Inheritance, Simple Refactoring, Encapsulation Approach

Structural Design Patterns

- Focus on the composition and organization of classes and objects.
- Ensure that if you modify one part of a system
  - Has minimal impact on other parts
- Enabling more flexible and efficient design.

Creational Design Pattern (Recap)

- Focus is object creation.

- Examples:
  - Singleton: Ensures one instance of a class.
  - Factory Method: Creates objects via a common interface.
  - Abstract Factory: Creates families of related objects.
  - Builder: Constructs complex objects step-by-step.
  - Prototype: Clones existing objects.

- Key Aspect: Abstracts and simplifies object instantiation.

Structural Design Patterns

- Adapter: Converts one interface to another compatible one.
- Facade: Simplifies a complex subsystem with a unified interface.
- Bridge: Separates abstraction from implementation.
- Composite: Treats individual and grouped objects uniformly.
- Decorator: Adds behavior to objects dynamically.
- Flyweight: Shares data to reduce memory usage.
- Proxy: Controls access or adds functionality as a substitute.

Adapter Design Pattern

- Useful when you have an existing class with a different interface than what your application expects.

- To make them compatible
  - The adapter pattern "adapts" the old interface to the new one
  - Acting as a bridge between the two.

Adapter Design Pattern

- An enterprise software system that generates reports and needs to print them.
- New system expects to use a standardized printing interface (e.g., `print()`)
- Organization has been using an old legacy printer
  - With its own proprietary interface (`printOld()`).

Adapter Design Pattern

- New software cannot directly use the old printer's interface because of incompatibility.

Incompatability

```
// Old interface
class OldPrinter {
public:
    void printOld() {
        cout << "Printing with Old Printer" << endl;
    }
};

// New interface expected by the client
class Printer {
public:
    virtual void print() = 0;
};
```

Incompatability

```
15. // Client code
16. int main() {
17.     OldPrinter oldPrinter;
18.     oldPrinter.printOld();  // This works, but it is not compatible with the
'Printer' interface.
19.
20.     // Client expects to call 'print()' on a 'Printer' interface, but
'OldPrinter' has 'printOld()'.
21.     Printer* printer;
22.     printer = &oldPrinter; // ERROR: Incompatible types, cannot assign
OldPrinter to Printer*
23.
24.     return 0;
25. }
26.
```

Simple Refactoring Approach

```
1. #include <iostream>
2. using namespace std;
3.
4. // Refactored old printer to match the new interface
5. class Printer {
6. public:
7.     virtual void print() = 0;
8. };
9.
10. class OldPrinter : public Printer {
11. public:
12.     void print() override {
13.         cout << "Printing with Refactored Old Printer" << endl;
14.     }
15. };
16.
17. int main() {
18.     Printer* printer = new OldPrinter();
19.     printer->print(); // Works directly without adaptation
20.     delete printer;
21.     return 0;
22. }
23.
```

Simple Refactoring Approach

- Modifies `OldPrinter`, may be impossible for legacy/third-party/closed-source.
- Code changes might introduce bugs or compatibility issues.
- Could disrupt system dependencies on old behavior/interface.

Multiple Inheritance Approach

```
 1. #include <iostream>
 2. using namespace std;
 3.
 4. // Old interface
 5. class OldPrinter {
 6. public:
 7.     void printOld() {
 8.         cout << "Printing with Old Printer" << endl;
 9.     }
10. };
11.
12. // New interface
13. class Printer {
14. public:
15.     virtual void print() = 0;
16. };
17.
```

Multiple Inheritance Approach

```
18. // Multiple inheritance approach
19. class PrinterAdapter : public Printer, public OldPrinter {
20. public:
21.     void print() override {
22.         printOld(); // Directly calls old method
23.     }
24. };
25. 
26. int main() {
27.     Printer* printer = new PrinterAdapter();
28.     printer->print(); // Works like an adapter
29.     delete printer;
30.     return 0;
31. }
32. 
17. 
```

Multiple Inheritance Approach

- Tight Coupling: Adapter depends on both interfaces
- Multiple inheritance risks conflicts if methods share names
- Code becomes harder to maintain, especially in complex systems.

Encapsulation Approach

```
1. #include <iostream>
2. using namespace std;
3.
4. class OldPrinter {
5. public:
6.     void printOld() {
7.         cout << "Printing with Old Printer" << endl;
8.     }
9. };
10.
11. class NewPrinter {
12. private:
13.     OldPrinter oldPrinter; // Encapsulating the old printer
14. public:
15.     void print() {
16.         oldPrinter.printOld(); // Call to the old method
17.     }
18. };
```

Encapsulation Approach

```
19.
20. // Client
21. int main() {
22.     NewPrinter printer;
23.     printer.print(); // Works, but directly tied to old implementation
24.     return 0;
25. }
26.
```

Encapsulation Approach

- Tight Coupling: `NewPrinter` depends on `OldPrinter`, reducing flexibility.
- No Full Adaptation: Wraps old method but doesn't match expected interface.
- Limited Polymorphism: Restricts replacing/extending `NewPrinter` with other implementations.

Adapter Design Pattern

```
 1. #include <iostream>
 2. using namespace std;
 3.
 4. // Old interface: represents an incompatible class
 5. class OldPrinter {
 6. public:
 7.     // This method is different from the expected interface
 8.     void printOld() {
 9.         cout << "Printing with Old Printer" << endl;
10.     }
11. };
12.
```

Adapter Design Pattern

```
13. // New interface: the client expects this interface
14. class Printer {
15. public:
16.     // Abstract method for printing (expected by the client)
17.     virtual void print() = 0;
18. };
19.
```

Adapter Design Pattern

```
20. // Adapter class: makes OldPrinter compatible with the new interface
21. class PrinterAdapter : public Printer {
22. private:
23.     OldPrinter* oldPrinter;  // Holds a reference to the old printer
24.
25. public:
26.     // Constructor that takes an OldPrinter object
27.     PrinterAdapter(OldPrinter* p) : oldPrinter(p) {}
28.
29.     // Implements the new interface
30.     void print() override {
31.         // The adapter calls the old interface method internally
32.         oldPrinter->printOld();
33.         // This allows us to use the old functionality in a new way
34.     }
35. };
36.
```

Adapter Design Pattern

```
37. // Client code: works with the new interface (Printer)
38. int main() {
39.     // Create an instance of the old printer
40.     OldPrinter* oldPrinter = new OldPrinter();
41.
42.     // Create an adapter that makes OldPrinter behave like a Printer
43.     Printer* printer = new PrinterAdapter(oldPrinter);
44.
45.     // Client uses the new interface to print
46.     printer->print();  // Outputs: Printing with Old Printer
47.
48.     // Clean up memory
49.     delete oldPrinter;
50.     delete printer;
51.
52.     return 0;
```

Adapter Design Pattern

- Old Class (`OldPrinter`)
  - Has `printOld()` method, not compatible with the client's expected interface.

- New Interface (`Printer`)
  - Abstract interface with a print() method, expected by the client.

Adapter Design Pattern

- Adapter Class (`PrinterAdapter`)
  - Implements Printer interface.
  - Holds a reference to `OldPrinter`.
  - Converts `print()` calls to `printOld()` internally.
  - Enables client to use `OldPrinter` via the new interface.

1. Compatibility with New Systems

- Smooth migration, seamless compatibility, minimal business disruption.
- Old printer has a different interface.
- Introduced new `Printer` interface.
- Integrates old system with new without code changes.

2. Consistency Across the System

- Business Standardization: Ensures consistent system behavior.
- Uniform Interface: All printers align with Printer for uniformity.
- Direct `OldPrinter` Use: Conflicts with business standards.
- Business Impact: Inconsistencies, integration risks, potential process failures.

3. Future Scalability and Flexibility

- Scalability and flexibility for future changes.
- Direct `OldPrinter` Use limits adding new printers; client code changes needed.
- Adapter enables easy integration of new printers.
- No client code changes, aligns with open-closed principle.

4. Compliance and Regulations

- Regulatory Requirements: Standardized interfaces needed for compliance.
- Direct `OldPrinter` use risks non-compliance.
- Adapter ensures adherence to new, compliant interface.
- Supports auditability, security, and industry standards.

5. Minimizing Risk

- `OldPrinter` is fragile, error-prone.
- Direct use has a risk of client code issues or bugs.
- Adapter reduces risk, keeps old functionality intact.
- Risk mitigation essential in finance, healthcare; avoid downtime

6. Gradual Migration Strategy

- Gradual migration needed
- Adapter enables interaction without full overhaul.
- Avoids immediate, costly disruption.
- Maintains functionality during transition.

7. Third-Party Integration Constraints

- `OldPrinter` is vendor-specific, non-modifiable.
- Adapter enables use within the new system.
- Aligns with internal standards and requirements

8. Separation of Concerns

- Separation of Concerns: Client focuses on core tasks (e.g., print jobs).
- Compatibility handling is managed as an infrastructure concern.
- Improves maintainability, aligns with business architecture policies.

9. Cost and Efficiency

- Adapting is cheaper than rewriting legacy systems.
- Rewriting: High costs, downtime, and risks.
- Adapter is conomical, keeps legacy systems functional.
- Meets Business Needs

Scenario: Multi-Provider Payment Gateway

- Your organization manages a global e-commerce platform that supports multiple payment methods: credit cards, digital wallets, and bank transfers.
- Each payment provider (e.g., Stripe, PayPal, and local banks) has its own API and interface, leading to inconsistencies in how they handle payment processing.
- Direct integration with each provider is not feasible due to differences in API signatures, authentication methods, and data formats.
- Design a system to use a standardized interface, ensuring uniform interactions, regardless of the provider.

Requirements

- Seamless payment integration
- Scalable for new providers
- Preserve existing client code
- Maintain core business logic

Example Adapter Design Pattern

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

using namespace std;

// Standardized interface for payment processors
class PaymentProcessor {
public:
    // Initializes payment for a given user
    virtual void initializePayment(const string& user) = 0;

    // Executes payment for a specified amount
    virtual void executePayment(double amount) = 0;

    // Refunds a specified amount
    virtual void refundPayment(double amount) = 0;
};
```

Example Adapter Design Pattern

```
// Old interface: Stripe API
class StripeAPI {
public:
    // Starts a transaction for a given user
    void startTransaction(const string& user) {
        cout << "Stripe: Starting transaction for " << user << endl;
    }

    // Charges the specified amount
    void charge(double amount) {
        cout << "Stripe: Charging $" << amount << endl;
    }

    // Refunds the specified amount
    void refund(double amount) {
        cout << "Stripe: Refunding $" << amount << endl;
    }
};
```

Example Adapter Design Pattern

```
// Old interface: PayPal API
class PayPalAPI {
public:
    // Makes a payment for a user and amount
    void makePayment(const string& user, double amount) {
        cout << "PayPal: Making payment of $" << amount << " for " << user <<
endl;
    }

    // Reverses the payment of a specified amount
    void reversePayment(double amount) {
        cout << "PayPal: Reversing payment of $" << amount << endl;
    }
};
```

Example Adapter Design Pattern

```
// Old interface: Bank Transfer API
class BankTransferAPI {
public:
    // Processes a bank transfer for an account and amount
    void processBankTransfer(const string& account, double amount) {
        cout << "Bank: Processing bank transfer of $" << amount << " for account "
<< account << endl;
    }

    // Reverses a bank transfer of the specified amount
    void reverseTransfer(double amount) {
        cout << "Bank: Reversing bank transfer of $" << amount << endl;
    }
};
```

Example Adapter Design Pattern

```
// Adapter for Stripe API
class StripeAdapter : public PaymentProcessor {
private:
    StripeAPI stripe; // Instance of the old Stripe API

public:
    // Initializes payment by starting a transaction
    void initializePayment(const string& user) override {
        stripe.startTransaction(user); // Calls old Stripe method
    }

    // Executes payment by charging the amount
    void executePayment(double amount) override {
        stripe.charge(amount); // Calls old Stripe method
    }

    // Refunds the specified amount
    void refundPayment(double amount) override {
        stripe.refund(amount); // Calls old Stripe method
    }
};
```

Example Adapter Design Pattern

```
// Adapter for PayPal API
class PayPalAdapter : public PaymentProcessor {
private:
    PayPalAPI paypal; // Instance of the old PayPal API

public:
    // Initializes payment for the user
    void initializePayment(const string& user) override {
        // PayPal requires preparation for execution in separate methods
        cout << "PayPal: Preparing payment for " << user << endl;
    }

    // Executes payment using PayPal's makePayment method
    void executePayment(double amount) override {
        paypal.makePayment("User", amount); // Calls old PayPal method
    }

    // Refunds the specified amount via PayPal's reversePayment
    void refundPayment(double amount) override {
        paypal.reversePayment(amount); // Calls old PayPal method
    }
};
```

Example Adapter Design Pattern

```
// Adapter for Bank Transfer API
class BankTransferAdapter : public PaymentProcessor {
private:
    BankTransferAPI bank; // Instance of the old Bank Transfer API

public:
    // Prepares a bank transfer for the user
    void initializePayment(const string& user) override {
        cout << "Bank: Preparing bank transfer for user " << user << endl;
    }

    // Executes bank transfer for a given amount
    void executePayment(double amount) override {
        bank.processBankTransfer("UserAccount", amount); // Calls old bank method
    }

    // Refunds the bank transfer amount
    void refundPayment(double amount) override {
        bank.reverseTransfer(amount); // Calls old bank method
    }
};
```

Example Adapter Design Pattern

```
// Client function to process an order
void processOrder(PaymentProcessor* processor, const string& user, double amount)
{
    // Step 1: Initialize the payment for the user
    processor->initializePayment(user);

    // Step 2: Execute the payment for the specified amount
    processor->executePayment(amount);

    // (Optional) Step 3: Refund could be added here, if needed
}
```

Example Adapter Design Pattern

```
int main() {
    // Example 1: Using Stripe as the payment processor
    PaymentProcessor* stripePayment = new StripeAdapter(); // Adapter for Stripe
    processOrder(stripePayment, "Alice", 100.00); // Processes payment via Stripe

    // Example 2: Using PayPal as the payment processor
    PaymentProcessor* paypalPayment = new PayPalAdapter(); // Adapter for PayPal
    processOrder(paypalPayment, "Bob", 200.00); // Processes payment via PayPal

    // Example 3: Using Bank Transfer as the payment processor
    PaymentProcessor* bankPayment = new BankTransferAdapter(); // Adapter for Bank Transfer
    processOrder(bankPayment, "Charlie", 300.00); // Processes payment via Bank Transfer

    // Clean up allocated memory
    delete stripePayment;
    delete paypalPayment;
    delete bankPayment;

    return 0;
}
```

Conclusion

- Structural Design Patterns Overview
- Adapter Design Pattern
- Benefits and Limitations
- Adapter Design Pattern vs
  - Multiple Inheritance, Simple Refactoring, Encapsulation Approach