Lecture 2

Dr. Umair Rehman

Agenda

- Encapsulation
- Why coupling matters in OOP (dependencies shape flexibility).
- Data variation → use parameters, not subclasses.
- Behavioral variation → use interfaces (abstractions).
- Four dependency patterns:
  - Concrete+Instantiate
  - Concrete+Inject,
  - Abstract+Instantiate
  - Abstract+Inject (DIP).
- Principle: Couple to abstractions, not concretions.
- Case studies

Encapsulation

- Core concept of Object-Oriented Programming (OOP),
- Bundling the data (attributes) and methods (functions) that operate on the data into a single unit or class
- Restricting access to some of the object's components
- Internal state of an object is protected
  - Direct access
  - Modified by other objects

Key Aspects of Encapsulation

- Data Hiding:
  - Using access modifiers like `private`, `protected`, and `public`
  - Control who can access the class attributes and methods

Key Aspects of Encapsulation

- Private members are accessible only within the class they are declared

- Protected members are accessible within the class and by derived classes

- Public members are accessible from outside the class

Key Aspects of Encapsulation

- Getter and Setter Methods:
  - Directly accessing the data, encapsulation promotes using
  - *Getter and setter methods* to read and modify the values of private members

- This allows for additional validation or logic before changes are made

Getter and Setter Methods

```
class Person {
private:
    string name;  // Private data member

public:
    // Getter method to access the private member 'name'
    string getName() {
        return name;
    }

    // Setter method to modify the private member 'name'
    void setName(string newName) {
        name = newName;
    }
};

```

Benefits of Encapsulation

- Security:
  - Prevents unintended interference or manipulation of the object's data
- Maintainability:
  - Internal implementation of a class can change
  - Without affecting other parts of the code
  - Public interface remains the same
- Reusability

Example of Encapsulation

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

class BankAccount {
private:
    double balance;  // Private attribute, can't be accessed directly from outside

public:
    // Constructor
    BankAccount(double initial_balance) {
        if (initial_balance >= 0)
            balance = initial_balance;
        else
            balance = 0;
    }
```

Example of Encapsulation

```
    // Getter method for balance (read-only access)
    double getBalance() {
        return balance;
    }

    // Setter method to deposit amount (write access with validation)
    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    // Setter method to withdraw amount (write access with validation)
    void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }
};
```

Example of Encapsulation

```cpp

int main() {
    BankAccount account(1000);  // Initial balance of 1000

    // Access balance through the public interface
    cout << "Initial Balance: $" << account.getBalance() << endl;

    account.deposit(500);  // Deposit 500
    cout << "After Deposit: $" << account.getBalance() << endl;

    account.withdraw(200);  // Withdraw 200
    cout << "After Withdrawal: $" << account.getBalance() << endl;

    return 0;
}

```

Example of Encapsulation:

- The `balance` attribute is private
- Can only be accessed through the public methods (`getBalance`, `deposit`, `withdraw`)
- Ensures that any changes to the balance are controlled and validated through these methods

Conceptual Integrity

- Successful systems should have conceptual integrity
- A consistent and unified vision across the entire project
- Source controls the design decisions for the system to maintain consistency
- System’s complexity is hidden behind a simpler, unified interface
  - As a class in OOP hides its internal workings behind public methods

Conceptual Integrity

- Isolates the core complexity of the design
- Rest of the team to focus on implementation
- Do not worry about the design decisions

Conceptual Integrity

- Modularization
    - Breaking down large systems into smaller, manageable modules
    - Similar to how classes and objects encapsulate complexity in OOP
    - Each module have a well-defined interface, keeping internal details hidden

Coupling Choices in OOP: From Concretions to Abstractions

- Concrete + Instantiate → class creates and owns a concrete type
  - Tightest coupling, least flexible
- Concrete + Inject → class receives a concrete type from outside
  - Avoids creation, but still tied to concretion
- Abstract + Instantiate → class depends on an abstraction, but instantiates concrete inside
  - Gains polymorphism, still rigid
- Abstract + Inject → class depends only on abstraction, concretes injected at runtime
  - Dependency Inversion Principle (DIP), most flexible/testable

1) Concrete + Instantiate (worst)

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

class Paypal {
public:
    void pay(double amount) {
        std::cout << "Paid $" << amount << " with Paypal\n";
    }
};

class Checkout {
    Paypal processor;   // OWNED concrete member (composition)
public:
    void process(double amount) {
        // METHOD CALL: calls Paypal::pay()
        processor.pay(amount);
    }
};

int main() {
    Checkout c;
    c.process(50.0);   // Hardwired to Paypal
}
```

1) Concrete + Instantiate (worst)

- Checkout hardcodes a concrete payment class and creates it inside.

2) ⚠️ Concrete + Inject

```cpp
class Paypal {
public:
    void pay(double amount) {
        std::cout << "Paid $" << amount << " with Paypal\n";
    }
};

class Checkout {
    Paypal& processor;  // Aggregation: holds reference
public:
    Checkout(Paypal& p) : processor(p) {}
    void process(double amount) {
        // METHOD CALL: calls Paypal::pay()
        processor.pay(amount);
    }
};

int main() {
    Paypal paypal;
    Checkout c(paypal);   // Inject concrete object
    c.process(75.0);
}
```

2) ⚠️ Concrete + Inject

- Checkout depends on the concrete Paypal class but gets it injected.

3) ⚠️ Abstract + Instantiate

```cpp
class IPaymentProcessor {
public:
    virtual ~IPaymentProcessor() = default;
    virtual void pay(double amount) = 0;
};

class Paypal : public IPaymentProcessor {
public:
    void pay(double amount) override {
        std::cout << "Paid $" << amount << " with Paypal\n";
    }
};

class Stripe : public IPaymentProcessor {
public:
    void pay(double amount) override {
        std::cout << "Paid $" << amount << " with Stripe\n";
    }
};
```

3) ⚠️ Abstract + Instantiate

```cpp

class Checkout {
    Paypal p1;   // Composition: creates concretes
    Stripe p2;
public:
    Checkout() : p1(), p2() {}
    void process(double amount) {
        // METHOD CALLS: both concretes
        p1.pay(amount/2);
        p2.pay(amount/2);
    }
};

int main() {
    Checkout c;
    c.process(100.0);
}

```

3) ⚠️ Abstract + Instantiate

- Introduce an abstraction (IPaymentProcessor) but Checkout still instantiates the concretes.

4) Abstract + Inject (best, DIP)

```
class IPaymentProcessor {
public:
    virtual ~IPaymentProcessor() = default;
    virtual void pay(double amount) = 0;
};

class Paypal : public IPaymentProcessor {
public:
    void pay(double amount) override {
        std::cout << "Paid $" << amount << " with Paypal\n";
    }
};

class Stripe : public IPaymentProcessor {
public:
    void pay(double amount) override {
        std::cout << "Paid $" << amount << " with Stripe\n";
    }
};

```

4) ✅ Abstract + Inject (best, DIP)

```cpp
class Checkout {
    IPaymentProcessor& processor;   // Aggregation: depends on abstraction
public:
    Checkout(IPaymentProcessor& p) : processor(p) {}
    void process(double amount) {
        // METHOD CALL: calls chosen concrete’s pay()
        processor.pay(amount);
    }
};

int main() {
    Paypal paypal;
    Stripe stripe;

    Checkout c1(paypal);   // inject Paypal
    Checkout c2(stripe);   // inject Stripe

    c1.process(120.0);   // "Paid $120 with Paypal"
    c2.process(80.0);    // "Paid $80 with Stripe"
}

```

4) ✅ Abstract + Inject (best, DIP)

- Checkout depends only on IPaymentProcessor and gets processors injected.

Summary

Style	What happens	Relationship
Concrete + Instantiate	Checkout creates a Paypal itself	Composition
Concrete + Inject	Checkout receives a Paypal	Aggregation
Abstract + Instantiate	Checkout depends on IPaymentProcessor but still news concretes	Composition
Abstract + Inject (best)	Checkout depends on IPaymentProcessor and gets concretes injected	Aggregation

Interfaces vs Parameters vs Concretes

- When to use what

Interfaces

- Use when variation is behavioral (different algorithms).
- Enables polymorphism and Dependency Inversion Principle (DIP).
- Consumers couple to the abstraction, not the concrete.Flexible, extensible, testable.

Parameters

- Use when variation is only data (e.g., numbers, labels, configs).
- Avoids subclass explosion for trivial differences.
- Keep one class, configure via constructor arguments or fields.
- Lightweight, clear.

Concretes

- Smallest/simple approach (no abstraction).
- Fine for tiny apps or one-off utilities.
- But consumers are tied to the concrete implementation.
- Harder to extend or swap behavior later.

Rule of Thumb

- Behavior → InterfaceData → ParameterToy/simple case → Concrete

1) Abstraction exists, but variation is only data

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

class ILogger {
public:
    virtual ~ILogger() = default;
    virtual void log(const std::string& msg) const = 0;
};

// Both print to console; only the prefix differs → this is *just data*.
class DebugLogger : public ILogger {
public:
    void log(const std::string& msg) const override {
        std::cout << "[DEBUG] " << msg << "\n";
    }
};

```

1) Abstraction exists, but variation is only data

```cpp
class ErrorLogger : public ILogger {
public:
    void log(const std::string& msg) const override {
        std::cout << "[ERROR] " << msg << "\n";
    }
};

int main() {
    DebugLogger d;  // pointless separate class
    ErrorLogger e;  // pointless separate class
    d.log("System booted.");
    e.log("Disk failure!");
}

```

Right way: one parameterized concrete under the same abstraction

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

class ILogger {
public:
    virtual ~ILogger() = default;
    virtual void log(const std::string& msg) const = 0;
};

// One class; the *level* is DATA (constructor arg).
class ConsoleLogger : public ILogger {
    std::string level;
public:
    explicit ConsoleLogger(std::string lvl) : level(std::move(lvl)) {}
    void log(const std::string& msg) const override {
        std::cout << "[" << level << "] " << msg << "\n";
```

Right way: one parameterized concrete under the same abstraction

```
};

int main() {
    ConsoleLogger debug("DEBUG");   // data = "DEBUG"
    ConsoleLogger error("ERROR");   // data = "ERROR"

    ILogger& l1 = debug;            // callers couple to abstraction
    ILogger& l2 = error;

    l1.log("System booted.");
    l2.log("Disk failure!");
}
```

Lesson

- When variation is only data (e.g., label text), don’t make subclasses. Keep the interface, use one parameterized class.

2) No abstraction; no subclasses for data variation

- Single concrete class with a data parameter. (Simple, but callers now couple to a concretion.)

2) No abstraction; no subclasses for data variation

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

class Logger {                                  // no interface here
    std::string level;
public:
    explicit Logger(std::string lvl) : level(std::move(lvl)) {}
    void log(const std::string& msg) const {
        std::cout << "[" << level << "] " << msg << "\n";
    }
};

```

2) No abstraction; no subclasses for data variation

```

int main() {
    Logger debug("DEBUG");         // data-only variation handled by ctor arg
    Logger error("ERROR");

    debug.log("Starting up...");
    error.log("Something went wrong!");
}

```

2) No abstraction; no subclasses for data variation

- Trade-off: Minimal and fine for tiny apps, but your code depends on a concrete type.
- If you later want a different logging behavior (e.g., write to file), you'll refactor callers.

3) With interfaces (behavioral variation + DI)

- Different algorithms (console vs file), so separate classes under an interface.
- Consumer couples to the abstraction and uses constructor injection.

3) With interfaces (behavioral variation + DI)

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

// Abstraction
class ILogger {
public:
    virtual ~ILogger() = default;
    virtual void log(const std::string& msg) const = 0;
};

// Concrete #1: Console behavior
class ConsoleLogger : public ILogger {
public:
    void log(const std::string& msg) const override {
        std::cout << "[CONSOLE] " << msg << "\n";
    }
};
```

3) With interfaces (behavioral variation + DI)

```cpp

// Concrete #2: File behavior (different algorithm)
class FileLogger : public ILogger {
    std::string filename;
public:
    explicit FileLogger(std::string fn) : filename(std::move(fn)) {}
    void log(const std::string& msg) const override {
        std::ofstream out(filename, std::ios::app);
        out << "[FILE] " << msg << "\n";
    }
};
```

3) With interfaces (behavioral variation + DI)

```
// High-level consumer depends on abstraction and gets a concrete via DI
class AuditService {
    ILogger& logger;                                 // injected dependency (aggregation)
public:
    explicit AuditService(ILogger& l) : logger(l) {}
    void record(const std::string& msg) { logger.log(msg); }  // polymorphic call
};
```

3) With interfaces (behavioral variation + DI)

```

int main() {
    ConsoleLogger console;
    FileLogger file("audit.log");

    AuditService a(console);          // inject console
    AuditService b(file);             // inject file

    a.record("User logged in.");
    b.record("Transaction completed.");
}

```

Examples

- Example 1: Keep abstraction, but handle data-only variation with one parameterized class (don't create subclasses for mere labels).

- Example 2: Smallest approach: no abstraction, one parameterized concrete (callers now depend on concretion).

- Example 3: When behavior differs, use interfaces + DI to couple to the abstraction and swap implementations freely.

Conclusion

- Encapsulation
- Why coupling matters in OOP (dependencies shape flexibility).
- Data variation → use parameters, not subclasses.
- Behavioral variation → use interfaces (abstractions).
- Four dependency patterns:
  - Concrete+Instantiate
  - Concrete+Inject,
  - Abstract+Instantiate
  - Abstract+Inject (DIP).
- Principle: Couple to abstractions, not concretions.
- Case studies