Lecture 6

Dr. Umair Rehman

Agenda

- S O L I D principles
  - SRP
  - OCP
  - LSP
  - ISP
  - DIP

Single Responsibility Principle (SRP)

“A class should have only one reason to change.”

- A module should encapsulate one axis of change or responsibility.
- Responsibility here means a stakeholder or concern (business logic, persistence, presentation, etc.).
- If a class has multiple reasons to change, it couples unrelated concerns and becomes fragile.

SRP Violation Example

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

class Report {
    std::string content;
public:
    Report(const std::string& text) : content(text) {}

    // Business logic: generate a report
    void generate() {
        std::cout << "Generating Report: " << content << std::endl;
    }

    // Persistence: save the report to a file
    void saveToFile(const std::string& filename) {
        std::ofstream file(filename);
        file << content;
        file.close();
    }
```

SRP Violation Example

```cpp
    // Communication: send report via email
    void sendEmail(const std::string& address) {
        std::cout << "Sending report to " << address << std::endl;
        // Pretend SMTP logic is here...
    }
};

int main() {
    Report report("Quarterly Sales");
    report.generate();
    report.saveToFile("sales.txt");
    report.sendEmail("ceo@company.com");
}

```

Why This Breaks SRP

- Multiple responsibilities in one class:
  - Report content generation (business logic).
  - File persistence (I/O).
  - Email communication (messaging).
- If file format changes, `Report` must change.
- If email system changes, `Report` must change.
- If business rules change, `Report` must also change.

Refactored Example

```cpp
#include <iostream>   // for console output
#include <fstream>    // for file output
#include <string>     // for std::string

// ===============================
// (1) Class: Report
// -------------------------------
// Responsibility: ONLY holds report content
// and can generate/return it.
// It does NOT know about saving or emailing.
// ===============================
class Report {
    std::string content;  // the "business data"
public:
    Report(const std::string& text) : content(text) {}

    // Business logic: generating report
    void generate() const {
        std::cout << "Generating Report: " << content << std::endl;
    }

    // Expose content safely (read-only)
    std::string getContent() const { return content; }
};

```

Refactored Example

```
// ================================
// (2) Class: ReportSaver
// --------------------------------
// Responsibility: handles persistence.
// It knows how to save a Report to disk,
// but it does NOT generate or email reports.
// ================================
class ReportSaver {
public:
    void saveToFile(const Report& report, const std::string& filename) const {
        std::ofstream file(filename);         // open file for writing
        file << report.getContent();          // save report content
        file.close();                         // close file
    }
};
```

Refactored Example

```
// ================================
// (3) Class: ReportSender
// --------------------------------
// Responsibility: handles communication.
// It knows how to send a Report somewhere,
// but it does NOT generate or save reports.
// ================================
class ReportSender {
public:
    void sendEmail(const Report& report, const std::string& address) const {
        std::cout << "Sending report to " << address << std::endl;
        std::cout << "Content: " << report.getContent() << std::endl;
        // Real SMTP logic would go here...
    }
};

```

Refactored Example

```cpp

// ================================
// (4) MAIN
// --------------------------------
// Demonstrates how these pieces interact.
// Notice: Report is passed to saver/sender.
// Relationships are ASSOCIATIONS, not inheritance.
// ================================
int main() {
    Report report("Quarterly Sales");

    report.generate();  // (1) Business logic only

    ReportSaver saver;
    saver.saveToFile(report, "sales.txt");  // (2) Persistence

    ReportSender sender;
    sender.sendEmail(report, "ceo@company.com"); // (3) Communication
}

```

Refactoring

- Each class has one reason to change:
  - If report content logic changes → only `Report`.
  - If saving format changes (CSV, JSON, DB) → only `ReportSaver`.
  - If communication method changes (email → Slack) → only `ReportSender`.

- Scalability:
  - You can add new saving strategies (`ReportDatabaseSaver`) or sending strategies (`ReportSlackSender`) without touching the original `Report`.

2. Open/Closed Principle (OCP)

“Software entities should be open for extension, but closed for modification.”

- You should be able to add new behavior (extension) without changing existing, stable code (modification).
- Achieved through abstraction (interfaces, base classes) or composition (strategy, plugins).

OCP Violation Example

```cpp
// OCP_VIOLATION.cpp
// g++ -std=c++17 OCP_VIOLATION.cpp -o ocp_bad && ./ocp_bad
// This file intentionally demonstrates an Open/Closed Principle violation.
// OCP says: "Software entities should be open for extension, but closed for modification."
// Below, we are NOT closed to modification: every new payment method forces edits
// inside PaymentProcessor::process(...), risking regressions and tight coupling.

#include <iostream>   // for std::cout
#include <string>     // for std::string
#include <stdexcept>  // for std::runtime_error

// ------------------------------
// Domain: A simple invoice
// ------------------------------
class Invoice {
    double amount_;                    // total amount due
    std::string customerEmail_;        // customer contact (for receipts)
public:
    // Construct with total amount and an email
    Invoice(double amount, std::string email)
        : amount_(amount), customerEmail_(std::move(email)) {}

    // Read-only getters (const correctness)
    double amount()        const { return amount_; }
    const std::string& email() const { return customerEmail_; }
};
```

OCP Violation Example

```cpp
};

// --------------------------------
// Low-level: fake utilities
// (Inline to keep the example self-contained.)
// --------------------------------
void saveReceiptToDisk(const std::string& text) {
    // Pretend to write to a file — we just print.
    std::cout << "[SAVE RECEIPT] " << text << "\n";
}

void sendEmail(const std::string& to, const std::string& body) {
    // Pretend to send an email — we just print.
    std::cout << "[EMAIL → " << to << "] " << body << "\n";
}

```

OCP Violation Example

```cpp
// ------------------------------
// Anti-OCP smell: an enum of "types"
// Adding a new method (e.g., ApplePay) means editing this enum
// AND editing the switch in PaymentProcessor::process(...).
// ------------------------------
enum class PaymentMethod {
    CreditCard,   // handle CC fees, gateways, etc. Named constnts for enumeraion
    PayPal        // handle PayPal fees, API quirks, etc.
    // ApplePay, BankTransfer, Crypto... (each addition will require code changes below)
};

// ------------------------------
// Bad design: One monolithic processor branching on type.
// This is a HIGH-LEVEL policy class that directly depends on LOW-LEVEL details,
// and it must be MODIFIED whenever we add or tweak a payment method.
// ------------------------------
```

OCP Violation Example

```cpp
class PaymentProcessor {
public:
    // Process an invoice using a specific PaymentMethod.
    // VIOLATION: This method must be edited for every new method or fee rule change.
    void process(const Invoice& inv, PaymentMethod method) {
        // Start by logging a generic message
        std::cout << "[PROCESS] amount $" << inv.amount() << "\n";

        // Anti-OCP hotspot: big switch with method-specific behavior.
        switch (method) {
            case PaymentMethod::CreditCard: {
                // --- CreditCard-specific logic (detail) ---
                // Fee calculation logic tightly inlined here:
                // (what if fee rules change per card network or region?)
                const double feeRate = 0.029;            // 2.9% fee
                const double fixedFee = 0.30;            // $0.30 per transaction
                double total = inv.amount() * (1.0 + feeRate) + fixedFee;
```

OCP Violation Example

```
        // Call a "gateway" (pretend) — more inline detail:
         std::cout << "[CC] Charging via Stripe-like gateway...\n";
         std::cout << "[CC] Amount + fees = $" << total << "\n";

         // Produce a receipt here (policy mixed with detail):
         std::string receipt = "CC receipt for $" + std::to_string(total);
         saveReceiptToDisk(receipt);                 // tight coupling to storage
         sendEmail(inv.email(), receipt);            // tight coupling to comms
         break;
     }
```

OCP Violation Example

```
            case PaymentMethod::PayPal: {
                // --- PayPal-specific logic (detail) ---
                // Different fee structure; more magic numbers baked in:
                const double feeRate = 0.034;            // 3.4% fee
                const double fixedFee = 0.49;            // $0.49 per transaction
                double total = inv.amount() * (1.0 + feeRate) + fixedFee;

                // Call a "PayPal API" (pretend):
                std::cout << "[PP] Charging via PayPal-like API...\n";
                std::cout << "[PP] Amount + fees = $" << total << "\n";

                // Another receipt path duplicated:
                std::string receipt = "PayPal receipt for $" + std::to_string(total);
                saveReceiptToDisk(receipt);              // duplicated coupling
                sendEmail(inv.email(), receipt);         // duplicated coupling
                break;
             }
             default:
                // If someone passes a value we don't handle, blow up at runtime.
                // Another smell: fragile, error-prone branching.
                throw std::runtime_error("Unsupported payment method");
        }

        // If business rules change (e.g., tax, discounts, fraud checks),
        // we edit THIS function again. This grows into a "god method" over time.
    }
};
```

OCP Violation Example

```cpp

// ------------------------------
// MAIN: shows pressure to modify
// ------------------------------
int main() {
    // Create two invoices with different customers.
    Invoice a{100.00, "alice@example.com"};
    Invoice b{250.00, "bob@example.com"};

    // Create the (bad) processor.
    PaymentProcessor processor;

    // Process with CreditCard: executes CC branch
    processor.process(a, PaymentMethod::CreditCard);

    // Process with PayPal: executes PayPal branch
    processor.process(b, PaymentMethod::PayPal);

    // Now imagine product asks:
    //  - "Add ApplePay with a different fee table."
    //  - "Log JSON receipts too."
    //  - "Route EU cards to a different gateway."
    //
    // Each request forces MODIFICATIONS in PaymentProcessor::process(...),
    // violating OCP and increasing regression risk.
    return 0;
}
```

Why this clearly violates OCP

- High-level policy (`PaymentProcessor`) must be modified for every new payment method or rule change.
- Logic is branched by enum (`switch`/`if-else` ladder), ensuring constant churn.
- Details leak in (fees, gateways, receipts), tangling responsibilities and coupling.
- Testing becomes harder (big method, many paths).

Refactored Example

```
// OCP_REFACTORED.cpp
// g++ -std=c++17 OCP_REFACTORED.cpp -o ocp_good && ./ocp_good
#include <iostream>
#include <memory>
#include <string>
#include <vector>

// ------------------------------
// Domain: invoice (unchanged, simple DTO)
// ------------------------------
class Invoice {
    double amount_;
    std::string customerEmail_;
public:
    Invoice(double amount, std::string email)
        : amount_(amount), customerEmail_(std::move(email)) {}
    double amount() const { return amount_; }
    const std::string& email() const { return customerEmail_; }
};
```

Refactored Example

```cpp
// ------------------------------
// Cross-cutting helpers (kept tiny for demo)
// In a real system these would be separate services.
// ------------------------------
inline void saveReceipt(const std::string& text) {
    std::cout << "[SAVE RECEIPT] " << text << "\n";
}
inline void emailReceipt(const std::string& to, const std::string& text) {
    std::cout << "[EMAIL → " << to << "] " << text << "\n";
}

```

Refactored Example

```
// ------------------------------
// Abstraction: "what it means to process a payment"
// High-level code depends on this interface (DIP friendly).
// ------------------------------
struct IPaymentMethod {
    virtual ~IPaymentMethod() = default;
    virtual std::string name() const = 0;                  // for logging/metrics
    virtual std::string charge(const Invoice& inv) const = 0; // returns receipt text
};

// ------------------------------
// Concrete strategies (details). Each has ONE reason to change.
// Adding a new method = add a new class. No edits to the processor.
// ------------------------------
class CreditCardPayment : public IPaymentMethod {
public:
    std::string name() const override { return "CreditCard"; }
    std::string charge(const Invoice& inv) const override {
        constexpr double feeRate = 0.029; // 2.9%
        constexpr double fixed    = 0.30;
        double total = inv.amount() * (1.0 + feeRate) + fixed;
        std::cout << "[CC] Charge via Stripe-like gateway. Total $" << total << "\n";
        return "CC receipt $" + std::to_string(total);
    }
};
```

Refactored Example

```cpp
class PayPalPayment : public IPaymentMethod {
public:
    std::string name() const override { return "PayPal"; }
    std::string charge(const Invoice& inv) const override {
        constexpr double feeRate = 0.034; // 3.4%
        constexpr double fixed    = 0.49;
        double total = inv.amount() * (1.0 + feeRate) + fixed;
        std::cout << "[PP] Charge via PayPal-like API. Total $" << total << "\n";
        return "PayPal receipt $" + std::to_string(total);
    }
};
```

Refactored Example

```cpp
// Example of adding a new method WITHOUT touching the processor:
class ApplePayPayment : public IPaymentMethod {
public:
    std::string name() const override { return "ApplePay"; }
    std::string charge(const Invoice& inv) const override {
        constexpr double feeRate = 0.020; // pretend cheaper
        constexpr double fixed    = 0.10;
        double total = inv.amount() * (1.0 + feeRate) + fixed;
        std::cout << "[APAY] Charge via ApplePay gateway. Total $" << total << "\n";
        return "ApplePay receipt $" + std::to_string(total);
    }
};
```

Refactored Example

```cpp
// ------------------------------
// High-level policy: processes payments but depends ONLY on IPaymentMethod.
// OCP: Closed to modification; Open to extension (new strategies).
// ------------------------------
class PaymentProcessor {
public:
    void process(const Invoice& inv, const IPaymentMethod& method) const {
        std::cout << "[PROCESS] $" << inv.amount()
                  << " via " << method.name() << "\n";
        const std::string receipt = method.charge(inv);  // polymorphic dispatch
        saveReceipt(receipt);
        emailReceipt(inv.email(), receipt);
        // Note: If receipt policy/storage changes, refactor these concerns
        // into interfaces too—still without modifying payment strategies.
    }
};
```

Refactored Example

```cpp

// ------------------------------
// MAIN: show extension without modification
// ------------------------------
int main() {
    Invoice a{100.00, "alice@example.com"};
    Invoice b{250.00, "bob@example.com"};
    Invoice c{180.00, "cindy@example.com"};

    PaymentProcessor processor;

    CreditCardPayment cc;
    PayPalPayment     pp;
    ApplePayPayment   apay; // new method drops in, zero changes to processor

    processor.process(a, cc);
    processor.process(b, pp);
    processor.process(c, apay);
}

```

Refactoring

- `PaymentProcessor` depends on `IPaymentMethod` (an abstraction).
- To add ApplePay/Crypto/BankTransfer, you add a new class implementing `IPaymentMethod`.
- You do not modify `PaymentProcessor` (high-level policy stays closed to change).
- If you later want pluggable receipts/email/storage, introduce `IReceiptSink` / `INotifier` and inject them—still no changes to existing strategies or the processor.

3. Liskov Substitution Principle (LSP)

"Subtypes must be substitutable for their base types without altering the correctness of the program."

- A derived class must honor the contract of its base class.
- Clients using the base type should not need to know the concrete subtype to function correctly.
- Violations often occur when derived classes throw, restrict, or weaken behavior promised by the base type.

LSP Violation Example

```cpp
// LSP_BIRD_VIOLATION.cpp
// g++ -std=c++17 LSP_BIRD_VIOLATION.cpp -o lsp_bird_bad && ./lsp_bird_bad
#include <iostream>
#include <stdexcept>

// ------------------------------
// Base class: Bird
// ------------------------------
class Bird {
public:
    virtual ~Bird() = default;

    // Base contract: all birds can fly
    virtual void fly() {
        std::cout << "Flapping wings and flying high!\n";
    }
};

```

LSP Violation Example

```cpp

// -------------------------------
// Derived class: Penguin
// -------------------------------
// LSP violation: Penguins are birds,
// but they cannot fly. Substituting Penguin
// breaks the expectations of Bird::fly().
class Penguin : public Bird {
public:
    void fly() override {
        // Violation: contract says "bird can fly",
        // but Penguin breaks it with an exception.
        throw std::logic_error("Penguins cannot fly!");
    }
};

```

LSP Violation Example

```cpp
// ------------------------------
// Client code expecting any Bird
// ------------------------------
void makeItFly(Bird& b) {
    std::cout << "Client: I expect a bird to fly...\n";
    b.fly(); // <-- safe for most birds, not for penguins
}

int main() {
    Bird sparrow;
    Penguin penguin;

    std::cout << "Testing Sparrow:\n";
    makeItFly(sparrow);   //  works fine

    std::cout << "\nTesting Penguin:\n";
    try {
        makeItFly(penguin); //  breaks substitutability
    } catch (const std::logic_error& e) {
        std::cout << "Error: " << e.what() << "\n";
    }
}
```

Why is this an LSP Violation?

- The base class `Bird` promises: "you can always call `fly()`."
- Anywhere a `Bird` is expected, substituting a `Penguin` breaks the client's expectations.
- This means `Penguin` is not truly substitutable for `Bird` → LSP violation.

Refactored Example

```cpp
// LSP_BIRD_REFACTORED.cpp
// g++ -std=c++17 LSP_BIRD_REFACTORED.cpp -o lsp_good && ./lsp_good
#include <iostream>
#include <memory>
#include <vector>

// ------------------------------
// Core idea: model *capabilities*, not taxonomy assumptions.
// - Base "Bird" has only behavior true for *all* birds.
// - Flying/Swimming are separate interfaces (capabilities).
// - Sparrow implements IFlyable; Penguin implements ISwimmable.
// Clients depend on the capability they actually need.
// ------------------------------

// Capability: can fly
struct IFlyable {
    virtual ~IFlyable() = default;
    virtual void fly() = 0;
};

// Capability: can swim
struct ISwimmable {
    virtual ~ISwimmable() = default;
    virtual void swim() = 0;
};
```

Refactored Example

```cpp
// Base type for all birds: contains only universally valid behavior
class Bird {
public:
    virtual ~Bird() = default;
    virtual void display() const = 0; // e.g., print species
    // Note: NO fly() here – not all birds can fly.
};

// A flying bird: Sparrow
class Sparrow final : public Bird, public IFlyable {
public:
    void display() const override { std::cout << "Sparrow\n"; }
    void fly() override { std::cout << "Sparrow: flapping and flying!\n"; }
};

// A non-flying bird that swims: Penguin
class Penguin final : public Bird, public ISwimmable {
public:
    void display() const override { std::cout << "Penguin\n"; }
    void swim() override { std::cout << "Penguin: torpedo swimming!\n"; }
};

```

Refactored Example

```cpp
// ----- Client code that depends on *capabilities*, not Bird itself -----

// Client that needs *anything that can fly*.
// LSP-safe: any IFlyable works (Sparrow, Eagle, Duck...), and
// types that can’t fly (Penguin) simply don’t implement IFlyable.
void launchIntoSky(IFlyable& f) {
    std::cout << "Launching into sky...\n";
    f.fly();
}

// Client that needs *anything that can swim*.
void sendIntoWater(ISwimmable& s) {
    std::cout << "Sending into water...\n";
    s.swim();
}

```

Refactored Example

```
int main() {
    Sparrow sparrow;
    Penguin penguin;

    // Common behavior via Bird
    sparrow.display();
    penguin.display();

    // Capability-based use
    launchIntoSky(sparrow);   //  Works: Sparrow implements IFlyable
    // launchIntoSky(penguin); //  Compile-time error: Penguin can't fly

    sendIntoWater(penguin);   //  Works: Penguin implements ISwimmable
    // sendIntoWater(sparrow); //  Compile-time error: Sparrow can't swim

    return 0;
}
```

Refactoring

- No broken promises: `Bird` doesn't claim "all birds can fly."
- Substitutability is preserved: any `IFlyable` can be used where a flyer is required; non-flyers aren't even type-compatible, preventing misuse at compile time.
- Clients depend on abstractions (capabilities), not concrete classes or leaky base contracts.

4. Interface Segregation Principle (ISP)

"Clients should not be forced to depend upon interfaces they do not use."

- Prefer many small, role-specific interfaces over large, "fat" ones.
- This reduces coupling and makes it impossible to misuse an object by calling methods it doesn't truly support.
- Prevents classes from being burdened with no-op or error-throwing implementations.

ISP Violation Example

```cpp
// ISP_VIOLATION.cpp
// g++ -std=c++17 ISP_VIOLATION.cpp -o isp_bad && ./isp_bad
//
// ISP says: "Clients should not be forced to depend upon interfaces they do not use."
// This example creates a fat interface that forces lightweight devices/clients
// to implement or link against methods they don't need.

#include <iostream>
#include <stdexcept>
#include <string>

// ------------------------------------------------------------
// ❌ Fat interface: forces ALL implementers to support print/scan/fax.
// Many devices (or clients) only need a subset, but are forced to depend on all.
// ------------------------------------------------------------
struct IMultiFunctionDevice {
    virtual ~IMultiFunctionDevice() = default;
    virtual void print(const std::string& doc) = 0;
    virtual void scan(const std::string& dest) = 0;
    virtual void fax(const std::string& number) = 0;
};

```

ISP Violation Example

```cpp

// ------------------------------------------------------------
// A simple printer device: it can ONLY print.
// But because of the fat interface, it must "implement" scan() and fax() anyway.
// Typical outcomes: throw, no-op, logs saying "unsupported" → brittle APIs.
// ------------------------------------------------------------
class SimplePrinter : public IMultiFunctionDevice {
public:
    void print(const std::string& doc) override {
        std::cout << "[PRINT] " << doc << "\n";
    }
    void scan(const std::string& /*dest*/) override {
        // ❌ ISP violation symptom: device is forced to implement what it can't do.
        throw std::logic_error("SimplePrinter does not support scanning");
    }
    void fax(const std::string& /*number*/) override {
        // Another forced, meaningless implementation.
        throw std::logic_error("SimplePrinter does not support faxing");
    }
};

```

ISP Violation Example

```cpp

// ------------------------------------------------------------
// A scanner-fax combo: can scan and fax, but not print.
// Still forced to provide a print() implementation.
// ------------------------------------------------------------
class ScanFaxStation : public IMultiFunctionDevice {
public:
    void print(const std::string& /*doc*/) override {
        // ❌ Dead method: client might accidentally call this at runtime.
        throw std::logic_error("ScanFaxStation cannot print");
    }
    void scan(const std::string& dest) override {
        std::cout << "[SCAN] → saved to " << dest << "\n";
    }
    void fax(const std::string& number) override {
        std::cout << "[FAX]  → sent to " << number << "\n";
    }
};

```

ISP Violation Example

```cpp

// ---------------------------------------------------------------------
// Client that ONLY needs printing, but is coupled to the fat interface.
// If someone passes ScanFaxStation by mistake, runtime blows up.
// ---------------------------------------------------------------------
class PrintClient {
    IMultiFunctionDevice& device;  // ❌ depends on unused scan/fax too
public:
    explicit PrintClient(IMultiFunctionDevice& d) : device(d) {}

    void run() {
        // Client wants to print ONLY, but is forced to accept any IMultiFunctionDevice.
        device.print("Quarterly Report");
    }
};

```

ISP Violation Example

```

// ---------------------------------------------------------------
// Client that ONLY needs scanning, but still depends on print/fax.
// ---------------------------------------------------------------
class ScanClient {
    IMultiFunctionDevice& device;  // ❌ depends on print/fax unnecessarily
public:
    explicit ScanClient(IMultiFunctionDevice& d) : device(d) {}

    void run() {
        device.scan("scan.pdf");
    }
};

```

ISP Violation Example

```
int main() {
    SimplePrinter printer;     // supports only print
    ScanFaxStation sfs;        // supports scan + fax (no print)

    // Works: print-only client using a printer
    PrintClient printOnly(printer);
    printOnly.run();

    // Works: scan-only client using a scanner/fax device
    ScanClient scanOnly(sfs);
    scanOnly.run();

```

What makes this an ISP violation?

- A fat interface (`IMultiFunctionDevice`) forces implementers to provide methods they don’t support.
- Clients with narrow needs (print-only, scan-only) are coupled to unused members, increasing compile-time and runtime surface area for errors.
- Swapping implementations compiles (same interface) but fails at runtime when unsupported methods are called.

Refactored Example

```cpp
// ISP_REFACTORED.cpp
// g++ -std=c++17 ISP_REFACTORED.cpp -o isp_good && ./isp_good
#include <iostream>
#include <memory>
#include <string>

// -------- Role-specific interfaces (ISP) --------
struct IPrinter {
    virtual ~IPrinter() = default;
    virtual void print(const std::string& doc) = 0;
};

struct IScanner {
    virtual ~IScanner() = default;
    virtual void scan(const std::string& dest) = 0;
};

struct IFax {
    virtual ~IFax() = default;
    virtual void fax(const std::string& number) = 0;
};
```

Refactored Example

```cpp
// -------- Concrete devices implement ONLY what they support --------
class SimplePrinter final : public IPrinter {
public:
    void print(const std::string& doc) override {
        std::cout << "[PRINT] " << doc << "\n";
    }
};

class ScanFaxStation final : public IScanner, public IFax {
public:
    void scan(const std::string& dest) override {
        std::cout << "[SCAN]  -> " << dest << "\n";
    }
    void fax(const std::string& number) override {
        std::cout << "[FAX]   -> " << number << "\n";
    }
};

// Optional: a real multi-function device can implement multiple small interfaces.
class MultiFunctionPrinter final : public IPrinter, public IScanner, public IFax {
public:
    void print(const std::string& doc) override  { std::cout << "[MFP PRINT] " << doc << "\n"; }
    void scan(const std::string& dest) override  { std::cout << "[MFP SCAN]  -> " << dest << "\n"; }
    void fax(const std::string& number) override { std::cout << "[MFP FAX]   -> " << number << "\n"; }
};

```

Refactored Example

```
// -------- Clients depend ONLY on the capability they need --------
class PrintClient {
    IPrinter& printer;                              // Narrow dependency
public:
    explicit PrintClient(IPrinter& p) : printer(p) {}
    void run() { printer.print("Quarterly Report"); }
};

class ScanClient {
    IScanner& scanner;                              // Narrow dependency
public:
    explicit ScanClient(IScanner& s) : scanner(s) {}
    void run() { scanner.scan("scan.pdf"); }
};

```

Refactored Example

```cpp
class FaxClient {
    IFax& faxer;                            // Narrow dependency
public:
    explicit FaxClient(IFax& f) : faxer(f) {}
    void run() { faxer.fax("+1-555-0100"); }
};

// -------- Optional: an adapter/composer to assemble multi-function from parts --------
class CompositeMFD final : public IPrinter, public IScanner, public IFax {
    IPrinter& p; IScanner& s; IFax& f;    // Compose capabilities from separate devices
public:
    CompositeMFD(IPrinter& p_, IScanner& s_, IFax& f_) : p(p_), s(s_), f(f_) {}
    void print(const std::string& doc) override  { p.print(doc); }
    void scan(const std::string& dest) override  { s.scan(dest); }
    void fax(const std::string& num) override    { f.fax(num); }
};

```

Refactored Example

```
class FaxClient {
    IFax& faxer;                                // Narrow dependency
public:
    explicit FaxClient(IFax& f) : faxer(f) {}
    void run() { faxer.fax("+1-555-0100"); }
};

```

Refactored Example

```cpp

int main() {
    SimplePrinter       printer;          // prints only
    ScanFaxStation      scanfax;          // scans + faxes
    MultiFunctionPrinter mfp;             // all three

    // Clients wired to correct, minimal interfaces:
    PrintClient pc1(printer);  pc1.run();           // OK
    ScanClient  sc1(scanfax);  sc1.run();           // OK
    FaxClient   fc1(scanfax);  fc1.run();           // OK

    // Same clients with a true multi-function device:
    PrintClient pc2(mfp); pc2.run();
    ScanClient  sc2(mfp); sc2.run();
    FaxClient   fc2(mfp); fc2.run();

    // Compose a multi-function device from separate parts (adapter style):
    CompositeMFD composed(printer, scanfax, scanfax);
    PrintClient pc3(composed); pc3.run();
    ScanClient  sc3(composed); sc3.run();
    FaxClient   fc3(composed); fc3.run();

    // Compile-time safety (these lines would NOT compile if uncommented):
    // PrintClient bad1(scanfax);  // error: ScanFaxStation is not an IPrinter
    // ScanClient  bad2(printer);  // error: SimplePrinter is not an IScanner

    return 0;
}

```

Why this fixes ISP

- No fat interface: devices implement only the methods they actually support.
- Clients are narrow: `PrintClient` depends on `IPrinter` only, etc.
- Fewer runtime hazards: impossible to "accidentally" pass a non-printing device to a print client (it won't compile).
- Extensible: add new roles (e.g., `ICopier`) or new devices without breaking existing clients.
- Composable: `CompositeMFD` shows how to assemble capabilities from separate devices without creating a god interface.

5. Dependency Inversion Principle (DIP)

"Depend on abstractions, not on concretions."

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details; details (implementations) should depend on abstractions.
- Commonly realized through constructor injection and interfaces in C++, enabling easy substitution and testing.

DIP Violation Example

```cpp
// DIP_VIOLATION.cpp
// g++ -std=c++17 DIP_VIOLATION.cpp -o dip_bad && ./dip_bad
#include <iostream>
#include <string>

// ------------------------------
// Low-level detail #1: concrete email sender
// ------------------------------
class EmailService {
public:
    void send(const std::string& to, const std::string& msg) {
        std::cout << "[EMAIL -> " << to << "] " << msg << "\n";
    }
};

// ------------------------------
// Low-level detail #2: concrete SMS sender (unused here)
// (Shows how adding another transport would force changes upstream.)
// ------------------------------
class SmsService {
public:
    void send(const std::string& to, const std::string& msg) {
        std::cout << "[SMS   -> " << to << "] " << msg << "\n";
    }
};

```

DIP Violation Example

```cpp

// ------------------------------
// High-level policy: NotificationManager
// ❌ DIP VIOLATION:
//   - Depends on a concrete class (EmailService), not an abstraction.
//   - Creates the dependency itself (hard-wired construction).
//   - To support SMS/Slack/etc., you must EDIT this class.
// ------------------------------
class NotificationManager {
    EmailService email_; // ❌ Hard dependency on a low-level detail
public:
    // ❌ No way to inject a different sender (e.g., SmsService).
    //    The dependency is fixed at compile time.

    void notifyWelcome(const std::string& userContact) {
        // High-level policy decides to send a welcome message...
        const std::string msg = "Welcome aboard!";
        // ...but can only do it via EmailService because it's hard-coded:
        email_.send(userContact, msg); // ❌ High-level uses a concrete detail
    }

    void notifyAlert(const std::string& userContact, const std::string& alert) {
        email_.send(userContact, "ALERT: " + alert); // ❌ Same tight coupling
    }
};

```

DIP Violation Example

```
int main() {
    NotificationManager manager;                             // High-level module
    manager.notifyWelcome("alice@example.com");   // Works for email only
    manager.notifyAlert("bob@example.com", "CPU usage high");

    // Now product asks: "Support SMS or Slack."
    // With this design, you must:
    //   - Add SmsService/SlackService members here,
    //   - Add branching or new methods,
    //   - Recompile everything.
    // This is exactly what DIP tries to avoid.
    return 0;
}

```

Why this violates DIP (quick hits)

- High-level depends on low-level: `NotificationManager` knows about `EmailService` directly.
- No abstractions: there's no `IMessageService`/interface in between.
- No injection: the high-level module constructs its dependency, making it impossible to substitute at runtime or in tests.
- Every new transport (SMS/Slack) forces modifications to `NotificationManager` (ripple effects, recompiles, regressions).

Refactored Example

```
// DIP_REFACTORED.cpp
// g++ -std=c++17 DIP_REFACTORED.cpp -o dip_good && ./dip_good
#include <iostream>
#include <string>

// ------------------------------
// 1) Abstraction (the "contract")
// High-level modules depend on THIS, not on concrete details.
// ------------------------------
struct IMessageService {
    virtual ~IMessageService() = default;
    virtual void send(const std::string& to, const std::string& msg) = 0;
};

```

Refactored Example

```cpp
// ------------------------------
// 2) Low-level details implement the contract
// Each has a single reason to change (gateway/transport specifics).
// ------------------------------
class EmailService final : public IMessageService {
public:
    void send(const std::string& to, const std::string& msg) override {
        std::cout << "[EMAIL -> " << to << "] " << msg << "\n";
    }
};

class SmsService final : public IMessageService {
public:
    void send(const std::string& to, const std::string& msg) override {
        std::cout << "[SMS   -> " << to << "] " << msg << "\n";
    }
};
```

Refactored Example

```cpp

// -------------------------------
// 3) High-level policy depends ONLY on the abstraction.
// Dependency is supplied from the outside (constructor injection).
// -------------------------------
class NotificationManager {
    IMessageService& transport_; // reference to abstraction (no ownership here)
public:
    explicit NotificationManager(IMessageService& transport) : transport_(transport) {}

    void notifyWelcome(const std::string& userContact) {
        transport_.send(userContact, "Welcome aboard!");
    }
    void notifyAlert(const std::string& userContact, const std::string& alert) {
        transport_.send(userContact, "ALERT: " + alert);
    }
};

```

Refactored Example

```
// -------------------------------
// 4) Composition root (main) wires concrete details at the edges.
// Swapping transports requires NO changes to NotificationManager.
// -------------------------------
int main() {
    EmailService email;
    SmsService   sms;
    FakeService  fake; // pretend unit-test double

    // Use email
    NotificationManager viaEmail(email);
    viaEmail.notifyWelcome("alice@example.com");
    viaEmail.notifyAlert("ops@example.com", "CPU usage high");

    // Swap to SMS (no code changes to manager)
    NotificationManager viaSms(sms);
    viaSms.notifyWelcome("+1-555-0100");
    viaSms.notifyAlert("+1-555-0200", "Disk 95%");

    return 0;
```

Why this satisfies DIP

- High-level depends on abstractions (`IMessageService`), not concrete details.
- Details depend on the abstraction (Email/SMS implement `IMessageService`).
- Constructor injection decouples wiring from behavior (easy swapping, easy testing).

Conclusion

SOLID Principles — One-Line Teaching Notes

- SRP → Cohesion
- OCP → Extensibility
- LSP → Behavioral Substitutability
- ISP → Lean Contracts
- DIP → Dependency Flow Inversion