SLIIT UNI
THE KNOWLEDGE UNIVERSITY

# Sri Lanka Institute of Information Technology

## Faculty of Computing

SE1020 - Object-Oriented Programming

Year 01 and Semester 02

Lecture 07

# SOLID Principles

# Lecture Roadmap


1. Why Do We Need Design Principles?
2. Where Did SOLID Come From?
3. S — Single Responsibility Principle (SRP)
4. O — Open/Closed Principle (OCP)
5. L — Liskov Substitution Principle (LSP)
6. I — Interface Segregation Principle (ISP)
7. D — Dependency Inversion Principle (DIP)
8. Closing Summary
9. Quiz Questions
10. Mini Case Studies
11. Closing Thoughts

# What Happens as Software Grows?

- Teams change, requirements change, deadlines shift.
- Codebases that were once small grow into massive systems.
- Small mistakes in design create long-term maintenance nightmares.
- Systems become rigid, fragile, and expensive to change.

# What Happens as Software Grows?

- Teams change, requirements change, deadlines shift.
- Codebases that were once small grow into massive systems.
- Small mistakes in design create long-term maintenance nightmares.
- Systems become rigid, fragile, and expensive to change.

**Without good design:**

- Adding features becomes dangerous.
- Fixing bugs introduces new ones.
- Developers are afraid to touch old code!

# Symptoms of Bad Design

**Common indicators that design has decayed:**

- **Rigidity** – Difficult to change.
- **Fragility** – Easy to break.
- **Immobility** – Hard to reuse.
- **Viscosity** – Easier to hack than do it properly.
- **Opacity** – Hard to understand what the code does.

# Symptoms of Bad Design

**Common indicators that design has decayed:**

- **Rigidity** – Difficult to change.
- **Fragility** – Easy to break.
- **Immobility** – Hard to reuse.
- **Viscosity** – Easier to hack than do it properly.
- **Opacity** – Hard to understand what the code does.

Design rot is real – and SOLID helps fight it!

# Real-World Analogy: The Restaurant Kitchen

Imagine a restaurant kitchen:

- All the ingredients are thrown into one big fridge.
- Recipes are hand-written differently every day.
- Everyone shouts at each other when they need an ingredient.

**What happens?**

- Meals take forever.
- Orders are wrong.
- Customers leave unhappy.

# Real-World Analogy: The Restaurant Kitchen

Imagine a restaurant kitchen:

- All the ingredients are thrown into one big fridge.
- Recipes are hand-written differently every day.
- Everyone shouts at each other when they need an ingredient.

**What happens?**

- Meals take forever.
- Orders are wrong.
- Customers leave unhappy.

**In software, bad design is like a messy kitchen.**

# Good Kitchen, Good Code

In a professional kitchen:

- Ingredients are labeled and stored in correct stations.
- Recipes are standardized.
- Chefs know their roles: grill, salad, pastry.

# Good Kitchen, Good Code

In a professional kitchen:

- Ingredients are labeled and stored in correct stations.
- Recipes are standardized.
- Chefs know their roles: grill, salad, pastry.

**Similarly:**

- Good code has clear organization.
- Responsibilities are divided.
- Everyone (and every object) knows their role.

# Good Kitchen, Good Code

In a professional kitchen:

- Ingredients are labeled and stored in correct stations.
- Recipes are standardized.
- Chefs know their roles: grill, salad, pastry.

**Similarly:**

- Good code has clear organization.
- Responsibilities are divided.
- Everyone (and every object) knows their role.

*SOLID principles = Recipes for building a clean kitchen for your code!*

# A Brief History of SOLID

- Introduced by **Robert C. Martin** (Uncle Bob) around 2000.
- Named and popularized by Michael Feathers.
- Built on decades of object-oriented design experience.
- Focused on writing code that is flexible, extensible, and maintainable.

# A Brief History of SOLID

- Introduced by **Robert C. Martin** (Uncle Bob) around 2000.
- Named and popularized by Michael Feathers.
- Built on decades of object-oriented design experience.
- Focused on writing code that is flexible, extensible, and maintainable.

**Today:**

- SOLID principles are a foundation for Agile development.
- Critical for Test-Driven Development (TDD), Design Patterns, and Clean Architecture.

# The Five SOLID Principles

**SOLID is an acronym:**

- **S** – Single Responsibility Principle
- **O** – Open/Closed Principle
- **L** – Liskov Substitution Principle
- **I** – Interface Segregation Principle
- **D** – Dependency Inversion Principle

# The Five SOLID Principles

**SOLID is an acronym:**

- **S** – Single Responsibility Principle
- **O** – Open/Closed Principle
- **L** – Liskov Substitution Principle
- **I** – Interface Segregation Principle
- **D** – Dependency Inversion Principle

*Let's dive deep into each one!*

# Single Responsibility Principle – Definition

**Official Definition:**

*"A class should have one, and only one, reason to change."*

# Single Responsibility Principle – Definition

**Official Definition:**

*"A class should have one, and only one, reason to change."*

**Who said it?** Robert C. Martin ("Uncle Bob"), Agile Software Development.

# Single Responsibility Principle – Definition

**Official Definition:**

*"A class should have one, and only one, reason to change."*

**Who said it?** Robert C. Martin ("Uncle Bob"), Agile Software Development. **Why?** Classes with multiple responsibilities are more fragile

and harder to maintain.

# SRP — Explained Simply

In simple words:

- Each class should only do **one job**.
- If a class handles many jobs, changing one job might break another.

# SRP — Explained Simply

In simple words:

- Each class should only do **one job**.
- If a class handles many jobs, changing one job might break another.

**Think:** One actor, one responsibility!

# Real-World Analogy: The Actor Principle

- In a theater play, each actor plays one character.
- Imagine if one actor played 5 characters — switching costumes constantly, confusing everyone!

# Real-World Analogy: The Actor Principle

- In a theater play, each actor plays one character.
- Imagine if one actor played 5 characters — switching costumes constantly, confusing everyone!

**In Code:**

- A class should be responsible for just one role.
- Not mix unrelated responsibilities together.

# Bad Example: Employee Class

```
class Employee {
    void calculatePay() { ... }
    void saveToDatabase() { ... }
    void generateReport() { ... }
}
```

# Bad Example: Employee Class

```
class Employee {
    void calculatePay() { ... }
    void saveToDatabase() { ... }
    void generateReport() { ... }
}
```

**Problem:**

- Business logic (salary calculation)
- Persistence logic (database save)
- Presentation logic (report generation)

All mixed together!

# Problems When SRP is Violated

- **Tight Coupling:** Change in database affects business logic.
- **Higher Risk:** Bug fixes introduce unexpected side-effects.
- **Hard Testing:** Unit testing is complicated because responsibilities are tangled.
- **Hard Reuse:** You can't reuse parts without dragging unrelated code.

# Problems When SRP is Violated

- **Tight Coupling:** Change in database affects business logic.
- **Higher Risk:** Bug fixes introduce unexpected side-effects.
- **Hard Testing:** Unit testing is complicated because responsibilities are tangled.
- **Hard Reuse:** You can't reuse parts without dragging unrelated code.

SRP violation leads to fragile, tangled systems.

# How to Apply SRP

Split responsibilities into separate classes:

- **One class per role**.
- Clear boundaries of responsibility.
- Changes in one area don't affect others.

# How to Apply SRP

Split responsibilities into separate classes:

- **One class per role**.
- Clear boundaries of responsibility.
- Changes in one area don't affect others.

**Design Tip:** Whenever you see "and" in a class description, it probably needs a split!

# Good Example: Separate Classes

```
class Employee { ... }

class PayCalculator {
  double calculatePay(Employee e) { ... }
}

class EmployeeRepository {
  void save(Employee e) { ... }
}

class EmployeeReport {
  void generate(Employee e) { ... }
}
```

# Good Example: Separate Classes

```
class Employee { ... }

class PayCalculator {
  double calculatePay(Employee e) { ... }
}

class EmployeeRepository {
  void save(Employee e) { ... }
}

class EmployeeReport {
  void generate(Employee e) { ... }
}
```

Each class now has **one reason to change**.

# Checklist: Signs SRP Might Be Violated

- Class does many unrelated things.
- Class grows too large over time ("God Class").
- Class touches too many external systems (DB, UI, Email, etc.).
- Multiple teams need to modify the same class for different reasons.

# Checklist: Signs SRP Might Be Violated

- Class does many unrelated things.
- Class grows too large over time ("God Class").
- Class touches too many external systems (DB, UI, Email, etc.).
- Multiple teams need to modify the same class for different reasons.

When in doubt: Split it out!

# Mini-Quiz — SRP Practice

**Which classes violate SRP? (Select all that apply)**

1. A class that manages database connections and user sessions.
2. A class that only calculates invoice totals.
3. A class that reads files, writes to network, and processes payments.
4. A class that validates email formats.

# Mini-Quiz — SRP Practice

**Which classes violate SRP? (Select all that apply)**

1. A class that manages database connections and user sessions.
2. A class that only calculates invoice totals.
3. A class that reads files, writes to network, and processes payments.
4. A class that validates email formats.

**Answer:** 1 and 3 violate SRP.

# Open/Closed Principle — Definition

**Official Definition:**

> *"Software entities should be open for extension, but closed for modification."*

# Open/Closed Principle — Definition

**Official Definition:**

> *"Software entities should be open for extension, but closed for modification."*

**Meaning:**

- Add new behavior **without** changing existing code.
- Protect working code from being broken by changes.

# OCP — Explained Simply

In simple words:

- You should be able to extend the behavior of a class without altering its source code.
- Existing tested code should stay untouched as much as possible.

# OCP — Explained Simply

In simple words:

- You should be able to extend the behavior of a class without altering its source code.
- Existing tested code should stay untouched as much as possible.

**Think:** Build systems like LEGO blocks — add more pieces without reshaping old ones!

# Real-World Analogy: Power Socket Extensions

- Wall sockets are "closed" — you don't modify the wall wiring.
- But you can "extend" functionality — plug in extension cords, splitters, adapters.

# Real-World Analogy: Power Socket Extensions

- Wall sockets are "closed" — you don't modify the wall wiring.
- But you can "extend" functionality — plug in extension cords, splitters, adapters.

**In Code:**

- Core modules stay stable.
- New features plug in without breaking old ones.

# Bad Example: Graphic Editor with Switch Case

```
class GraphicEditor {
    void drawShape(Shape s) {
        if (s.type == "Circle") drawCircle(s);
        else if (s.type == "Square") drawSquare(s);
    }
}
```

# Bad Example: Graphic Editor with Switch Case

```
class GraphicEditor {
    void drawShape(Shape s) {
        if (s.type == "Circle") drawCircle(s);
        else if (s.type == "Square") drawSquare(s);
    }
}
```

Adding new shapes requires editing this method — violating OCP!

# Problems When OCP is Violated

- Fragile — changing switch logic may break unrelated shapes.
- Risky — one typo could crash the system.
- Inefficient — recompile and retest old working code every time.
- Bottleneck — one team must control all modifications.

# Problems When OCP is Violated

- Fragile — changing switch logic may break unrelated shapes.
- Risky — one typo could crash the system.
- Inefficient — recompile and retest old working code every time.
- Bottleneck — one team must control all modifications.

Bad for scalability and team growth.

# How to Apply OCP

Use **polymorphism**:

- Create an interface or abstract class.
- Let each new behavior implement or inherit.
- Core classes call the abstract methods — not care about details.

# How to Apply OCP

Use **polymorphism**:

- Create an interface or abstract class.
- Let each new behavior implement or inherit.
- Core classes call the abstract methods — not care about details.

This way, new behaviors come as new classes — no modification needed.

# Good Example: Using Polymorphism

```
interface Shape { void draw(); }

class Circle implements Shape {
  public void draw() { /* draw circle */ }
}

class Square implements Shape {
  public void draw() { /* draw square */ }
}

class GraphicEditor {
  void drawShape(Shape s) {
    s.draw();
  }
}
```

# Checklist: Signs OCP Might Be Violated

- Frequent modifications to stable code when adding new features.
- Growing long "if-else" or "switch-case" chains.
- Risk of introducing bugs in old logic when adding new options.
- New types require editing core processing classes.

# Checklist: Signs OCP Might Be Violated

- Frequent modifications to stable code when adding new features.
- Growing long "if-else" or "switch-case" chains.
- Risk of introducing bugs in old logic when adding new options.
- New types require editing core processing classes.

When in doubt: Abstract it out!

# Mini-Quiz — OCP Practice

**Which design follows OCP?**

1. A switch-case for every new payment method (Visa, MasterCard, PayPal).
2. A Payment interface with classes for Visa, MasterCard, PayPal implementing it.
3. A hardcoded if-else for each notification type (Email, SMS, Push).

# Mini-Quiz — OCP Practice

**Which design follows OCP?**

1. A switch-case for every new payment method (Visa, MasterCard, PayPal).
2. A Payment interface with classes for Visa, MasterCard, PayPal implementing it.
3. A hardcoded if-else for each notification type (Email, SMS, Push).

**Answer:** 2 follows OCP.

# Liskov Substitution Principle — Definition

**Official Definition:**

*"Subtypes must be substitutable for their base types without altering desirable behavior."*

# Liskov Substitution Principle — Definition

**Official Definition:**

> *"Subtypes must be substitutable for their base types without altering desirable behavior."*

**Introduced by:** Barbara Liskov, 1987.

# LSP — Explained Simply

In simple words:

- If code works with a base class, it should also work with any derived class — without surprises.
- Subclasses must honor the promises made by their parents.

# LSP — Explained Simply

In simple words:

- If code works with a base class, it should also work with any derived class — without surprises.
- Subclasses must honor the promises made by their parents.

**Think:** Child objects should behave like parents — or better.

# Real-World Analogy: Vehicle Rental

- You rent a "vehicle" expecting to drive.
- If the rental company gives you a "boat" instead, you're stuck!

# Real-World Analogy: Vehicle Rental

- You rent a "vehicle" expecting to drive.
- If the rental company gives you a "boat" instead, you're stuck!

**In Code:**

- Substituting types should not break client expectations.

# Bad Example: Rectangle and Square

```
class Rectangle {
  void setWidth(int w) {...}
  void setHeight(int h) {...}
}

class Square extends Rectangle {
  void setWidth(int w) { super.setWidth(w); super.setH
  void setHeight(int h) { super.setWidth(h); super.set
}
```

# Bad Example: Rectangle and Square

```
class Rectangle {
  void setWidth(int w) {...}
  void setHeight(int h) {...}
}

class Square extends Rectangle {
  void setWidth(int w) { super.setWidth(w); super.setH
  void setHeight(int h) { super.setWidth(h); super.set
}
```

Square "breaks" the behavior expected of Rectangle clients.

# Problems When LSP is Violated

- Clients get unpredictable behavior.
- Testing becomes complicated.
- Bugs are introduced silently — harder to detect.

# Problems When LSP is Violated

- Clients get unpredictable behavior.
- Testing becomes complicated.
- Bugs are introduced silently — harder to detect.

Violation destroys confidence in type hierarchy!

# How to Apply LSP

- Redesign the hierarchy carefully.
- Do not use inheritance if behavior cannot be preserved.
- Prefer interfaces or separate abstractions if substitution doesn't make sense.

# How to Apply LSP

- Redesign the hierarchy carefully.
- Do not use inheritance if behavior cannot be preserved.
- Prefer interfaces or separate abstractions if substitution doesn't make sense.

**Tip:** Inheritance = "is-a" relationship — check it carefully.

# Good Example: No Forced Inheritance

```
interface Shape { int area(); }

class Rectangle implements Shape {
  int width, height;
  int area() { return width * height; }
}

class Square implements Shape {
  int side;
  int area() { return side * side; }
}
```

# Good Example: No Forced Inheritance

```
interface Shape { int area(); }

class Rectangle implements Shape {
  int width, height;
  int area() { return width * height; }
}

class Square implements Shape {
  int side;
  int area() { return side * side; }
}
```

Both shapes implement the same contract without confusion.

# Checklist: Signs LSP Might Be Violated

- Subclass overrides methods and changes expected behavior.
- Subclass throws unexpected exceptions.
- Clients must add "instanceof" checks to distinguish subclasses.

# Checklist: Signs LSP Might Be Violated

- Subclass overrides methods and changes expected behavior.
- Subclass throws unexpected exceptions.
- Clients must add "instanceof" checks to distinguish subclasses.

When in doubt: Flatten the hierarchy!

# Mini-Quiz — LSP Practice

**Which one violates LSP?**

1. A Dog class extends Animal and behaves correctly.
2. A Duck class extends Bird but cannot fly (throws exception when fly() is called).
3. A Car class implements Driveable and drives safely.

# Mini-Quiz — LSP Practice

**Which one violates LSP?**

1. A Dog class extends Animal and behaves correctly.
2. A Duck class extends Bird but cannot fly (throws exception when fly() is called).
3. A Car class implements Driveable and drives safely.

**Answer:** 2 violates LSP.

# Interface Segregation Principle — Definition

**Official Definition:**

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

# Interface Segregation Principle — Definition

**Official Definition:**

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

**Meaning:**

- Prefer many small, focused interfaces over one large general-purpose interface.

# ISP — Explained Simply

In simple words:

- Clients should only know about the methods that are relevant to them.
- Don't burden classes with unnecessary obligations.

# ISP — Explained Simply

In simple words:

- Clients should only know about the methods that are relevant to them.
- Don't burden classes with unnecessary obligations.

**Think:** Don't make a printer implement "fax" if it can't fax!

# Real-World Analogy: Restaurant Menu

- Imagine a vegetarian ordering from a menu that forces them to choose a meat dish.
- Very confusing and annoying!

# Real-World Analogy: Restaurant Menu

- Imagine a vegetarian ordering from a menu that forces them to choose a meat dish.
- Very confusing and annoying!

**In Code:**

- Clients should only see the methods they need.

# Bad Example: Multi-Function Interface

```
interface Machine {
   void print();
   void scan();
   void fax();
}

class BasicPrinter implements Machine {
   public void print() { ... }
   public void scan() { throw new UnsupportedOperationE
   public void fax() { throw new UnsupportedOperationEx
}
```

# Bad Example: Multi-Function Interface

```
interface Machine {
   void print();
   void scan();
   void fax();
}

class BasicPrinter implements Machine {
   public void print() { ... }
   public void scan() { throw new UnsupportedOperationE
   public void fax() { throw new UnsupportedOperationEx
}
```

Classes are forced to implement methods they don't support!

# Problems When ISP is Violated

- Unnecessary code complexity.
- Higher risk of runtime errors (e.g., unsupported methods).
- Harder to understand and maintain.

# Problems When ISP is Violated

- Unnecessary code complexity.
- Higher risk of runtime errors (e.g., unsupported methods).
- Harder to understand and maintain.

Break interfaces into smaller, focused roles!

# How to Apply ISP

Split interfaces by responsibilities:

- One interface per logical group.
- Classes implement only what they actually support.

# How to Apply ISP

Split interfaces by responsibilities:

- One interface per logical group.
- Classes implement only what they actually support.

**Example:** Printer, Scanner, and Fax interfaces separately.

# Good Example: Focused Interfaces

```
interface Printer { void print(); }
interface Scanner { void scan(); }
interface Fax { void fax(); }

class BasicPrinter implements Printer {
  public void print() { ... }
}
```

Classes now only implement what they truly offer!

# Checklist: Signs ISP Might Be Violated

- Classes throw exceptions for unimplemented methods.
- Interfaces seem bloated or unrelated.
- Clients know too much about unrelated behavior.

# Checklist: Signs ISP Might Be Violated

- Classes throw exceptions for unimplemented methods.
- Interfaces seem bloated or unrelated.
- Clients know too much about unrelated behavior.

When in doubt: Split it out!

# Mini-Quiz — ISP Practice

**Which option follows ISP?**

1. A Device interface with print(), scan(), fax(), copy() methods.
2. Separate Printer, Scanner, Copier interfaces.

# Mini-Quiz — ISP Practice

**Which option follows ISP?**

1. A Device interface with print(), scan(), fax(), copy() methods.
2. Separate Printer, Scanner, Copier interfaces.

**Answer:** 2 follows ISP.

# Dependency Inversion Principle — Definition

**Official Definition:**

*"High-level modules should not depend on low-level modules. Both should depend on abstractions."*

# Dependency Inversion Principle — Definition

**Official Definition:**

*"High-level modules should not depend on low-level modules. Both should depend on abstractions."*

**Also:**

*"Abstractions should not depend on details. Details should depend on abstractions."*

# DIP — Explained Simply

In simple words:

- Code should depend on interfaces, not implementations.
- High-level logic (business rules) should not know how low-level parts (e.g., database, email) work.

# DIP — Explained Simply

In simple words:

- Code should depend on interfaces, not implementations.
- High-level logic (business rules) should not know how low-level parts (e.g., database, email) work.

**Think:** *"Don't hardcode dependencies. Inject flexibility."*

# Real-World Analogy: Universal Power Adapters

- Travelers use universal adapters — they work regardless of country.
- You plug into a standard interface, not directly into the wall's wiring.

# Real-World Analogy: Universal Power Adapters

- Travelers use universal adapters — they work regardless of country.
- You plug into a standard interface, not directly into the wall's wiring.

**In Code:**

- Design against interfaces, not hardcoded classes.

# Bad Example: Hardcoded Dependency

```
class EmailSender {
    void send(String message) { ... }
}

class OrderService {
    private EmailSender sender = new EmailSender();

    void completeOrder() {
        sender.send("Order completed");
    }
}
```

# Bad Example: Hardcoded Dependency

```
class EmailSender {
    void send(String message) { ... }
}

class OrderService {
    private EmailSender sender = new EmailSender();

    void completeOrder() {
        sender.send("Order completed");
    }
}
```

**Problem:**

- Can't reuse OrderService with a different message sender.
- Hard to test (no mocks).

# Problems When DIP is Violated

- Low-level changes ripple into high-level business logic.
- Difficult to replace components (e.g., swap database or logger).
- Harder to test in isolation.
- Leads to tight coupling between layers.

# Problems When DIP is Violated

- Low-level changes ripple into high-level business logic.
- Difficult to replace components (e.g., swap database or logger).
- Harder to test in isolation.
- Leads to tight coupling between layers.

Abstract away your dependencies!

# How to Apply DIP

**Solution:**

- Introduce an interface or abstract base class.
- Make high-level modules depend on that abstraction.
- Inject the concrete implementation at runtime (via constructor).

# How to Apply DIP

**Solution:**

- Introduce an interface or abstract base class.
- Make high-level modules depend on that abstraction.
- Inject the concrete implementation at runtime (via constructor).

**Use with:** IoC, Dependency Injection, Mocking for tests.

# Good Example: Abstracted Dependency

```
interface Notifier {
  void send(String message);
}

class EmailSender implements Notifier {
  public void send(String message) { ... }
}

class OrderService {
  private final Notifier notifier;

  OrderService(Notifier notifier) {
    this.notifier = notifier;
  }

  leteOrder() {
    notifier
```

# Good Example: Abstracted Dependency

```
interface Notifier {
  void send(String message);
}

class EmailSender implements Notifier {
  public void send(String message) { ... }
}

class OrderService {
  private final Notifier notifier;

  OrderService(Notifier notifier) {
    this.notifier = notifier;
  }

  leteOrder() {
    notifier
```

# Checklist: Signs DIP Might Be Violated

- Business logic instantiates low-level classes directly.
- Cannot easily switch implementations (e.g., File $\rightarrow$ DB $\rightarrow$ Cloud).
- Unit tests are difficult due to hardwired dependencies.
- Code breaks if low-level modules change their details.

# Checklist: Signs DIP Might Be Violated

- Business logic instantiates low-level classes directly.
- Cannot easily switch implementations (e.g., File $\rightarrow$ DB $\rightarrow$ Cloud).
- Unit tests are difficult due to hardwired dependencies.
- Code breaks if low-level modules change their details.

When in doubt: Invert the dependency!

# Mini-Quiz — DIP Practice

**Which example follows DIP?**

1. LoggerService creates a FileLogger directly in its constructor.
2. LoggerService accepts an ILogger interface via constructor.

# Mini-Quiz — DIP Practice

**Which example follows DIP?**

1. LoggerService creates a FileLogger directly in its constructor.
2. LoggerService accepts an ILogger interface via constructor.

**Answer:** 2 follows DIP.

# SOLID — Quick Recap

**S** — Single Responsibility Principle: Each class should have only one reason to change.

**O** — Open/Closed Principle: Classes should be open for extension, closed for modification.

**L** — Liskov Substitution Principle: Subtypes must be usable in place of their supertypes without altering behavior.

**I** — Interface Segregation Principle: Many client-specific interfaces are better than one general-purpose interface.

**D** — Dependency Inversion Principle: Depend on abstractions, not concretions.

# Final Key Messages

- Good design is deliberate, not accidental.
- SOLID principles are guidelines, not rigid rules.
- Always prioritize clarity, simplicity, and separation of concerns.
- Mastering SOLID leads to flexible, robust, and scalable systems.

# Final Key Messages

- Good design is deliberate, not accidental.
- SOLID principles are guidelines, not rigid rules.
- Always prioritize clarity, simplicity, and separation of concerns.
- Mastering SOLID leads to flexible, robust, and scalable systems.

*Design today what you'll be proud to maintain tomorrow!*

# Quick Quiz — Test Your Understanding

**Choose the correct answers:**

1. Which principle aims to minimize "God Classes"?

# Quick Quiz — Test Your Understanding

**Choose the correct answers:**

1. Which principle aims to minimize "God Classes"? (Answer: SRP)
2. Which principle says software should be extendable without modifying existing code?

# Quick Quiz — Test Your Understanding

**Choose the correct answers:**

1. Which principle aims to minimize "God Classes"? (Answer: SRP)
2. Which principle says software should be extendable without modifying existing code? (Answer: OCP)
3. Violating which principle leads to runtime surprises when substituting types?

# Quick Quiz — Test Your Understanding

**Choose the correct answers:**

1. Which principle aims to minimize "God Classes"? (Answer: SRP)
2. Which principle says software should be extendable without modifying existing code? (Answer: OCP)
3. Violating which principle leads to runtime surprises when substituting types? (Answer: LSP)
4. Which principle promotes using smaller, focused interfaces?

# Quick Quiz — Test Your Understanding

**Choose the correct answers:**

1. Which principle aims to minimize "God Classes"? (Answer: SRP)
2. Which principle says software should be extendable without modifying existing code? (Answer: OCP)
3. Violating which principle leads to runtime surprises when substituting types? (Answer: LSP)
4. Which principle promotes using smaller, focused interfaces? (Answer: ISP)
5. Which principle recommends depending on interfaces rather than implementations?

# Quick Quiz — Test Your Understanding

**Choose the correct answers:**

1. Which principle aims to minimize "God Classes"? (Answer: SRP)
2. Which principle says software should be extendable without modifying existing code? (Answer: OCP)
3. Violating which principle leads to runtime surprises when substituting types? (Answer: LSP)
4. Which principle promotes using smaller, focused interfaces? (Answer: ISP)
5. Which principle recommends depending on interfaces rather than implementations? (Answer: DIP)

# Discussion Questions

**Short Answer Discussion:**

- Give a real-world example where SRP violation caused a problem.
- How would OCP help in building a plugin system?
- How could LSP violations cause subtle bugs in polymorphic collections?

# Discussion Questions

**Short Answer Discussion:**

- Give a real-world example where SRP violation caused a problem.
- How would OCP help in building a plugin system?
- How could LSP violations cause subtle bugs in polymorphic collections?

# Mini Case Study — SRP

**Problem:**

- Class `InvoiceManager` handles invoice calculations, database saving, PDF generation, and emailing invoices.

**Task:**

- Identify how SRP is violated.
- Suggest how you would refactor the class.

# Mini Case Study — SRP

**Problem:**

- Class `InvoiceManager` handles invoice calculations, database saving, PDF generation, and emailing invoices.

**Task:**

- Identify how SRP is violated.
- Suggest how you would refactor the class.

**Hint:** One responsibility per class!

# Mini Case Study — OCP

**Problem:**

- A `NotificationService` uses a huge if-else ladder to send emails, SMS, and push notifications.

**Task:**

- How is OCP violated?
- How could you extend this system without modifying the service?

# Mini Case Study — OCP

**Problem:**

- A `NotificationService` uses a huge if-else ladder to send emails, SMS, and push notifications.

**Task:**

- How is OCP violated?
- How could you extend this system without modifying the service?

**Hint:** Think interfaces for each notification type.

# Mini Case Study — LSP

**Problem:**

- A `Bird` class has a `fly()` method. A `Penguin` subclass throws an exception when fly() is called.

**Task:**

- Why is LSP violated?
- How could you redesign this?

# Mini Case Study — LSP

**Problem:**

- A `Bird` class has a `fly()` method. A `Penguin` subclass throws an exception when fly() is called.

**Task:**

- Why is LSP violated?
- How could you redesign this?

**Hint:** Use capability-based interfaces (like `Flyable`).

# Mini Case Study — ISP

**Problem:**

- A `SmartDevice` interface includes call(), browseInternet(), playGames(), but a basic feature-phone only supports calling.

**Task:**

- How is ISP violated?
- How would you redesign the interfaces?

# Mini Case Study — ISP

**Problem:**

- A `SmartDevice` interface includes call(), browseInternet(), playGames(), but a basic feature-phone only supports calling.

**Task:**

- How is ISP violated?
- How would you redesign the interfaces?

**Hint:** Smaller, role-specific interfaces!

# Mini Case Study — DIP

**Problem:**

- An `OrderProcessor` creates a `MySQLDatabase` object directly inside its methods.

**Task:**

- Why does this violate DIP?
- Suggest a better design.

# Mini Case Study — DIP

**Problem:**

- An `OrderProcessor` creates a `MySQLDatabase` object directly inside its methods.

**Task:**

- Why does this violate DIP?
- Suggest a better design.

**Hint:** Depend on interfaces, inject dependencies!

# Thank You!

SOLID isn't a rulebook — it's a language for designing better systems.

Master SOLID — and you master the art of maintainable code!

(Questions? Discussion? Practical tips?)