Software Design Principles

UNSW SYDNEY

What Goes Wrong in Software Design?

- Initial design is clean and elegant, often well-structured.
- Over time, design degrades due to evolving requirements and rushed changes.
- Known as "software rot", this process makes code hard to maintain and evolve.

Symptoms:

- Rigidity: Small changes cause widespread impact.
- Fragility: One change breaks unrelated parts.
- Immobility: Useful components can’t be reused easily.
- Viscosity: Environment or process encourages hacks over clean design.

Rigidity and Fragility

Rigidity: System resists change due to interdependencies.

Example: A login module change forces updates in unrelated reporting or database modules.

Impact: Managers hesitate to allow even minor fixes.

Fragility: Changes result in unexpected breakages.

Example: Fixing an email validator crashes the profile picture upload feature.

Impact: Developer trust and morale drop; testing becomes difficult.

Observation**:** The above are due to poor dependency management, not just evolving requirements.

Immobility and Viscosity

Immobility: Modules can't be reused due to tight coupling.

> Example: A "*UserNotification*" class depends on web framework internals, so we cannot reuse in CLI app.

Design **viscosity**: Easier to do the wrong thing (hack) than the right thing.

Environmental **viscosity**: Long compile/test cycles encourage shortcuts.

> Example: Hack a feature with global variables instead of refactoring due to 20-minute build time.

Observation**:** Most symptoms of rot are caused by bad dependency structures.

What Are Software Design Principles?

- They provide guidelines to develop systems that are maintainable, flexible, reusable, and robust.
- Adhering to these principles helps to mitigate common software engineering issues such as design rot (degradation) and ensures software remains scalable and adaptable over time.
- Changing requirements don't *have to* ruin design.
- Good design anticipates change, however, bad design breaks under it.

Importance of Software Design Principles

- Maintainability: Software should be easy to update and enhance without extensive refactoring (re-engineering).
- Flexibility: Systems should adapt smoothly to changing requirements.
- Reusability: Components and modules should be designed to be easily reusable across various parts of the application or even in different projects.
- Robustness: The software should handle errors gracefully and maintain functionality under different circumstances.

SOLID Principles (1)

An acronym that represents five crucial principles for object-oriented design:

Single Responsibility Principle (SRP):

- A class should have only one reason to change, focusing on a single functionality.

Open/Closed Principle (OCP):

- Software entities should be open for extension but closed for modification.

Liskov Substitution Principle (LSP):

- Objects of a superclass should be replaceable with objects of subclasses without affecting the correctness of the program.

SOLID Principles (2)

Interface Segregation Principle (ISP):

- Clients should not be forced to depend on interfaces they do not use; favor many specific interfaces over a single general-purpose one.

Dependency Inversion Principle (DIP):

- Depend on abstractions, not concrete implementations. Higher-level modules should not depend on lower-level modules but rather on abstractions.

Why Follow These Principles?

- Preventing Software Rot: Avoid the deterioration of the software design over time.
- Ease of Maintenance: Reduce the cost and effort involved in updating and managing code.
- Enhanced Productivity: Developers spend less time debugging and refactoring, more on innovation and delivering value.
- Improved Collaboration: Clear, principle-driven design aids team communication and collaboration.

Real-world Example

Consider an online payment system:

Without design principles:

- Payment methods (Credit Card, PayPal, Crypto, etc.) tightly coupled in the codebase, making additions or modifications challenging and error-prone.

Applying SOLID principles:

- Each payment method is encapsulated within its class (SRP).
- Adding new payment methods requires implementing a payment interface without altering existing code (OCP, DIP).
- Users of payment classes aren't exposed to methods irrelevant to them (ISP).

Good Design

"Change in software is constant, good design embraces it!"

- Following structured design principles ultimately results in higher-quality, longer-lasting software.

Software Cohesion and Coupling

- Cohesion: The degree to which elements of a module/class belong together.
- Coupling: The degree of interdependence between software modules.
- High cohesion and low coupling are hallmarks of **good** software **design**.

What is Cohesion?

- Cohesion: The degree to which elements of a module/class belong together.
- High Cohesion: Elements of the module work towards a single purpose.
- Low Cohesion: Elements are unrelated or loosely related.
- Aim for high cohesion for maintainability and readability.

Examples of Cohesion

High Cohesion

Class: InvoiceProcessor

- Methods: calculateTotal(), applyDiscount(), generateInvoice()
- All methods related to processing an invoice.

Benefits: Easier to understand and maintain, Reusable

Low Cohesion

Class: UtilityClass

- Methods: readFile(), sendEmail(), sortArray()
- Functions unrelated to one another.

Problems: Hard to maintain, Difficult to test, Not reusable as a unit

What is Coupling?

- Coupling: The degree of interdependence between software modules.
- Tight Coupling: Modules heavily dependent on each other.
- Loose Coupling: Modules operate independently with minimal dependencies.
- Aim for loose coupling to enable flexibility and reuse.

Types of Coupling

Some of the important types of coupling are:

- Data Coupling: Modules share data through parameters.
- Control Coupling: One module controls the flow of another (e.g., flags).
- External Coupling: Modules depend on externally imposed data formats.
- Common Coupling: Shared global variables.
- Content Coupling: One module modifies data of another.

Examples of Coupling

Low Coupling

Modules: UserInterface, BusinessLogic, DataAccess

- Each layer interacts through interfaces.

Benefits: Easy to change or replace components, Improved testability

High Coupling

Class A calls methods of Class B directly and modifies its state.

Problems: Difficult to reuse or refactor, Ripple effects from changes

Design Tips for High Cohesion

- Use the Single Responsibility Principle (SRP), as far as possible.
- Group related functionalities.
- Avoid "God classes".
- Refactor when a class or method grows too large.

Design Tips for Low Coupling

- Minimize shared data
- Use interfaces and abstractions
- Apply Dependency Injection
- Use event-driven or observer patterns, for loosely dynamically coupled systems

When to Use Design Principles?

- Design principles help to remove design smells: needless complexity.
- However, they should not be used when there are no design smells.
- It is a mistake to blindly accept a principle just because it is one.
- Avoid over-adherence, it can create a new design smell: needless complexity.

Design Principle:

Principle of Least Knowledge (Law of Demeter)

- The Principle of Least Knowledge (also called the Law of Demeter) suggest that a module (or object) should only talk to its immediate "friends", and not to strangers.
- In simpler terms: “Only call methods on objects you directly know**.**”

- Formal Rule

  A method **M** of an object **O** may only invoke methods that belong to:

  1) O itself
  2) M's parameters
  3) Any objects created/instantiated within M
  4) O’s direct fields (its own instance variables)

Design Principle:

Principle of Least Knowledge (Law of Demeter)

- Minimises coupling: Prevents objects from becoming overly dependent on others' internal structure.
- Enhances maintainability: Changes in one class are less likely to ripple through the system.
- Improves encapsulation: Objects hide their data better and expose minimal necessary interfaces.

Code Example – Violating LoD (Tightly Coupled)

```
class Engine {
    public void start() { System.out.println("Engine started"); }
}

class Car {
    private Engine engine = new Engine();
    public Engine getEngine() { return engine; }
}

class Driver {
    public void startCar(Car car) {
        car.getEngine().start();
    }
}
```

Violates LoD,
accessing a “stranger” (engine)

Code Example – Respecting LoD (Loosely Coupled)

```
class Engine {
    public void start() { System.out.println("Engine started"); }
}

class Car {
    private Engine engine = new Engine();
    public void start() { engine.start(); }
}

class Driver {
    public void startCar(Car car) {
        car.start();
    }
}
```

Car mediates access

Talks only to its direct friend

Definition of LSP (Liskov Substitution Principle)

"Objects of a superclass should be replaceable with objects of a subclass without breaking the application."
- *Barbara Liskov, 1987*

- This ensures a subclass behaves in ways that do not surprise or violate the expectations set by the parent class.

- Formally:

  "Let S be a subtype of T. Then, objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program."

Real-World Analogy: LSP

- Superclass: **Bird**
  Subclass: **Penguin**

- Birds can fly, therefore fly() is in the base (super) class **Bird**.

- Penguins are birds, but they cannot fly.

- Problem: Substituting **Penguin** for **Bird** breaks the program!

Examples: LSP

Violating LSP

```java
class Bird {
    void fly() {
        System.out.println("Flying...");
    }
}

class Ostrich extends Bird {
    @Override
    void fly() {
        throw new UnsupportedOperationException("Ostrich can'
    }
}
```

Fixing the Violation

```java
interface Bird {
    void eat();
}

interface FlyingBird extends Bird {
    void fly();
}

class Sparrow implements FlyingBird {
    public void fly() { System.out.println("Sparrow flies"); }
    public void eat() { System.out.println("Sparrow eats"); }
}

class Ostrich implements Bird {
    public void eat() { System.out.println("Ostrich eats"); }
}
```

Example: LSP (Shape Hierarchy)

```
class Rectangle {
    int width, height;
    void setWidth(int w) { width = w; }
    void setHeight(int h) { height = h; }
    int area() { return width * height; }
}

class Square extends Rectangle {
    void setWidth(int w) {
        width = w;
        height = w;
    }
    void setHeight(int h) {
        width = h;
        height = h;
    }
}
```

We cannot substitute Square for Rectangle, may break logic expecting width != height.

Results in incorrect behavior!

```
Rectangle r1 = new Rectangle();
r1.setWidth(50); // only changes Width, as expected
r1 = new Square();
r1.setWidth(50); // Unexpectedly, it also changes Height!
```

So, we cannot replace an object of Rectangle by an object of Square!

Example: LSP (Shape Hierarchy)

```
class Rectangle {
    int width, height;
    void setWidth(int w) { width = w; }
    void setHeight(int h) { height = h; }
    int area() { return width * height; }
}

class Square extends Rectangle {
    void setWidth(int w) {
        width = w;
        height = w;
    }
    void setHeight(int h) {
        width = h;
        height = h;
    }
}
```

We cannot substitute Square for Rectangle, may break logic expecting width != height.

After refactoring

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

class Rectangle implements Shape { ... }
class Square implements Shape { ... }
```

Why LSP Matters

- Encourages correct hierarchy modelling
- Enables safe polymorphism
- Reduces unexpected behaviour at runtime
- Facilitates reusability and maintainability
- Think of LSP as a contract: subclasses must honour the guarantees of their parents.

Introduction to Covariance and Contravariance

- Covariance and Contravariance describe how types behave in inheritance when method overriding.
- Covariance: Return type can be more specific (subtype)
- Contravariance: Parameter types can be more general (supertype)

Covariant Return Types

- Allows the return type in an overridden method to be a subtype of the original.
- Enables more specific results while remaining compatible.

```
class Animal {}
class Dog extends Animal {}

class AnimalShelter {
    Animal adopt() { return new Animal(); }
}

class DogShelter extends AnimalShelter {
    @Override
    Dog adopt() { return new Dog(); }
}
```

Contravariance in Parameters

- Contravariant parameters accept supertypes of the original type.
- This is not allowed in typical method overriding (Java, C++).

```
class Parent {
    void process(Number n) { ... }
}

class Child extends Parent {
    void process(Integer i) { ... }
}
```

Not Overriding,
But results in Overloading!
Now there are two methods,
one each for *Number* and *Integer* types.

Rules Summary for Method Overriding

Aspect	Rule in OOP Overriding
Method Name	Must match
Parameters	Must be identical
Return Type	Covariant allowed
Exceptions	Can be narrower
Access Modifier	Can be more open

End