

**MASTERING SOFTWARE  
DEVELOPMENT**

A Guide For Early Career Engineers



Balwinder Sodhi

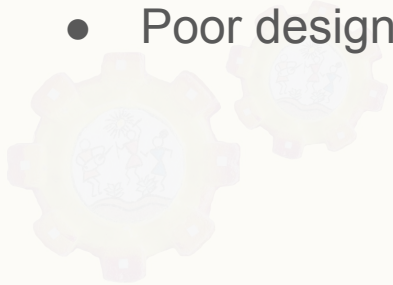
# Fundamentals of Software Design

Balwinder Sodhi

# Why Software Design Matters

- Shapes long-term maintainability, performance, extensibility, and developer productivity.
- Determines how easy it is to add features or fix bugs without breakage.
- Poor design compounds over time—teams eventually grind to a halt.

MASTERING  
SOFTWARE  
DEVELOPMENT  
A Guide For Early Career Engineers



Balwinder Sodhi

# Key Drivers

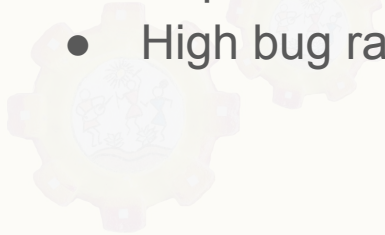
- **Changing Requirements**
  - Business priorities shift; a flexible architecture reduces rewrite cost.
- **System Complexity**
  - Clear design controls entropy and prevents unmanageable codebases.
- **Team Scale**
  - Good design enables multiple engineers to work without stepping on each other.
- **Quality Attributes (NFRs)**
  - Reliability, security, testability, performance, deployability.

Balwinder Sodhi

# Common Signs of Bad Design

- Simple changes requiring modification across 8–10 files.
- High coupling: modifying one module breaks others.
- Rigid classes with too many responsibilities.
- Duplicated logic scattered around the code.
- High bug rate in business-critical flows.

MASTERING  
SOFTWARE DEVELOPMENT  
A Guide For Early Career Engineers



Balwinder Sodhi

MASTERING SOFTWARE  
DEVELOPMENT

A Guide For Early Career Engineers



Balwinder Sodhi

# Core Software Design Principles

# Single Responsibility Principle (SRP)

- Essence of SRP:

- A module/class should have one reason to change.
- Reduces complexity and cognitive load.

- Pitfalls

- Misinterpreting SRP as “one method per class.”
- Creating too many micro-classes that add overhead without clarity.

**NEEDS  
WORK**



```
class UserService {  
    createUser(data) { ... }  
    sendWelcomeEmail(user) { ... }  
    generateUserReport(user) { ... }  
}
```



```
class UserCreator { ... }  
class WelcomeEmailSender { ... }  
class UserReportGenerator { ... }
```

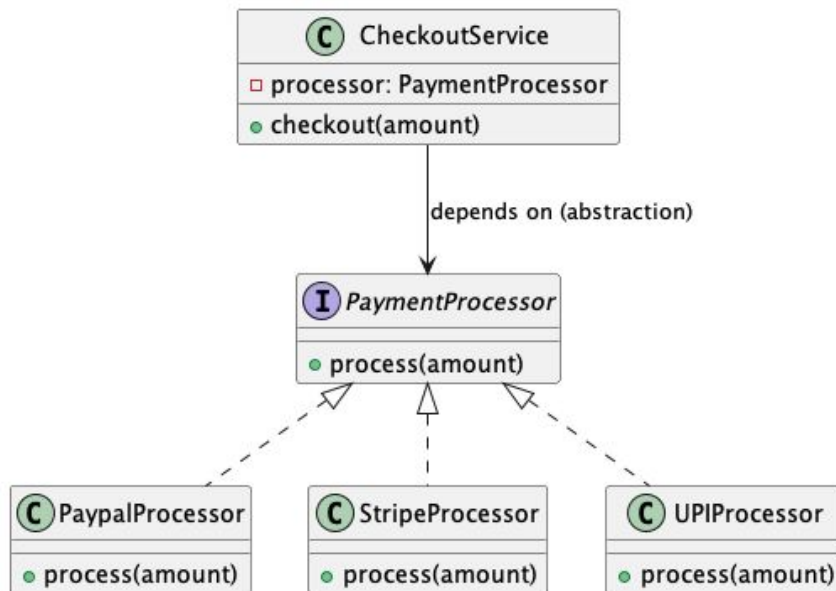
# Open/Closed Principle (OCP)

- Definition:

- Modules should be open for extension, closed for modification.
- Achieved using abstraction and polymorphism.

- Pitfalls:

- Overengineering abstraction layers “just in case”.
- Adding unnecessary design patterns prematurely.



# Liskov Substitution Principle (LSP)

- What LSP Really Means

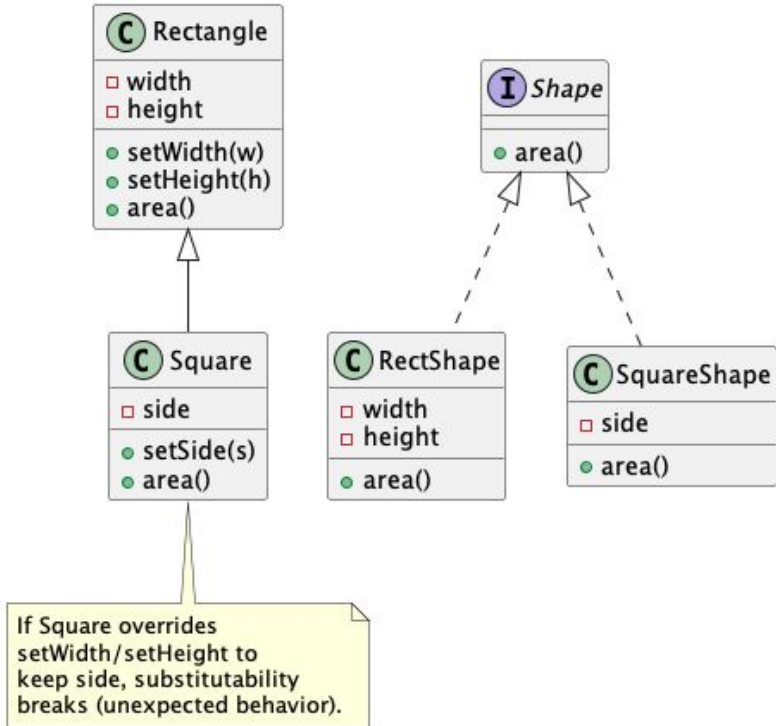
- Subtypes must behave as their base type promises.
- Violations appear when subclasses change expected behavior.

- Classic Violation

- Rectangle vs Square inheritance.
- Mutation semantics violate base expectations.

- Practical Tips

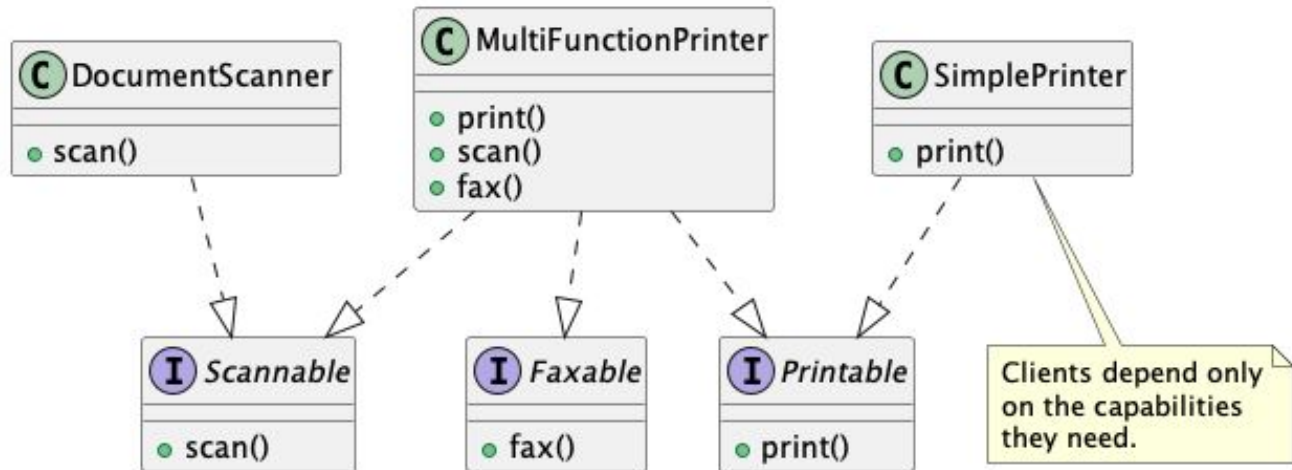
- Avoid inheritance when behavior diverges.
- Favor composition when substitutability feels forced.





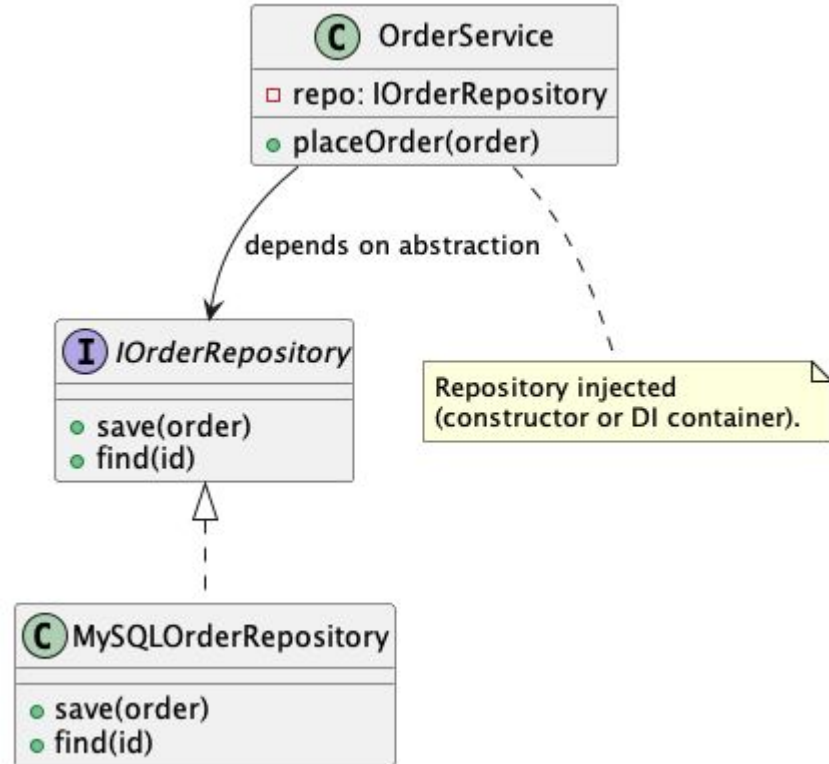
# Interface Segregation Principle (ISP)

- Essence: Clients shouldn't depend on interfaces they don't use.
- Pitfall: “God interfaces” that force unused methods on implementations.



# Dependency Inversion Principle (DIP)

- Essence:
  - High-level modules shouldn't depend on low-level details.
  - Both depend on abstractions.
- Pitfall: Creating too many meaningless interfaces (“IUserService1”, “IUserService2”).



# You Aren't Gonna Need It (YAGNI)

- Principle:

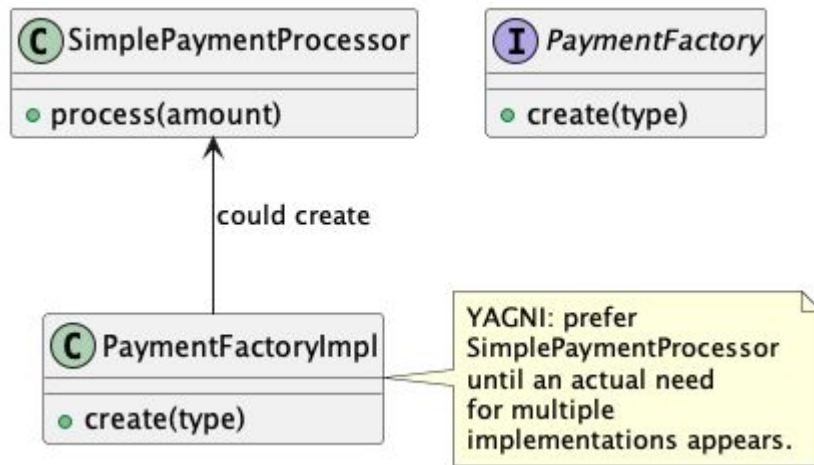
- Don't build abstractions or features until needed.
- Opposes speculative design.

- Where Engineers Violate YAGNI

- Adding extensibility points for hypothetical future use cases.
- Creating generalized frameworks for a single project.

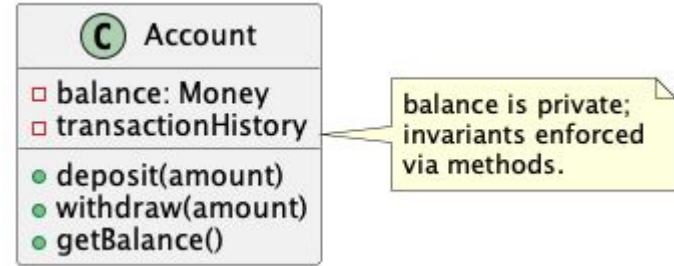
- Benefits

- Keeps codebase small, simple, and focused.
- Reduces maintenance cost by avoiding unused code paths.



# Encapsulation in Modern Software

- Hide internal complexity behind stable interfaces.
- Expose only what's necessary.
- Why It Matters
  - Prevents accidental misuse of internal state.
  - Enables safe refactoring without changing consumers.
- Pitfalls
  - Making everything public for “quick access”.
  - Abusing getters/setters instead of modeling real invariants.



# Don't Repeat Yourself (DRY)

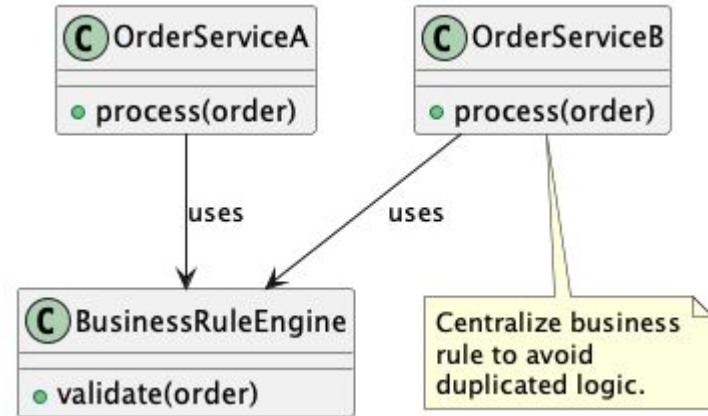
- Every piece of knowledge should have a single, authoritative representation.

- Types of Duplication

- Copy-paste duplication
  - Literal same code in two places.
- Knowledge duplication
  - Same business rule expressed differently across modules.

- Pitfalls

- Overusing DRY to merge unrelated logic.
- Creating generic utilities that become dumping grounds.



MASTERING SOFTWARE  
DEVELOPMENT

A Guide For Early Career Engineers



Balwinder Sodhi

# Introduction to Design Patterns

# Why Patterns Matter

- Provide proven structures to solve common design problems.
- Improve communication (“use an observer here”).
- Should be used pragmatically, not dogmatically.



# Singleton

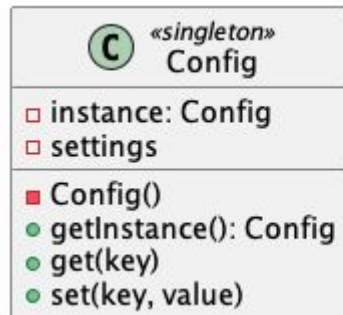
- Intent: Ensure a class has exactly one instance.

- Typical Uses

- Configuration loaders.
- Centralized resource managers.

- Pitfalls

- Global state → hard to test.
- Hidden coupling between modules.

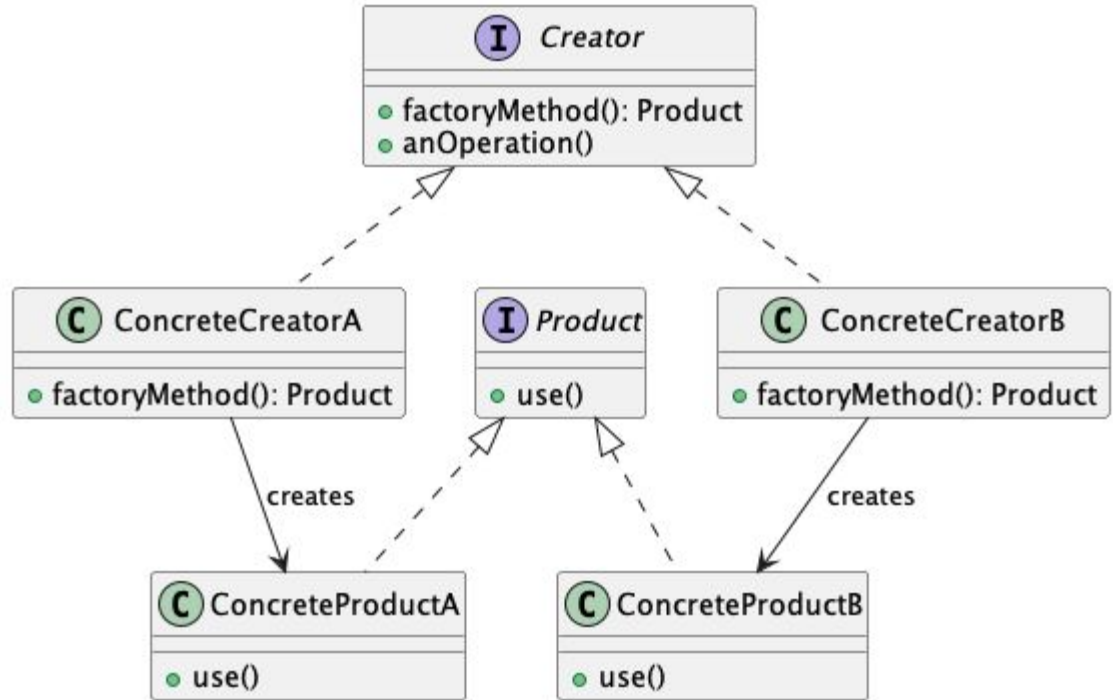


Use sparingly: global state makes testing harder.



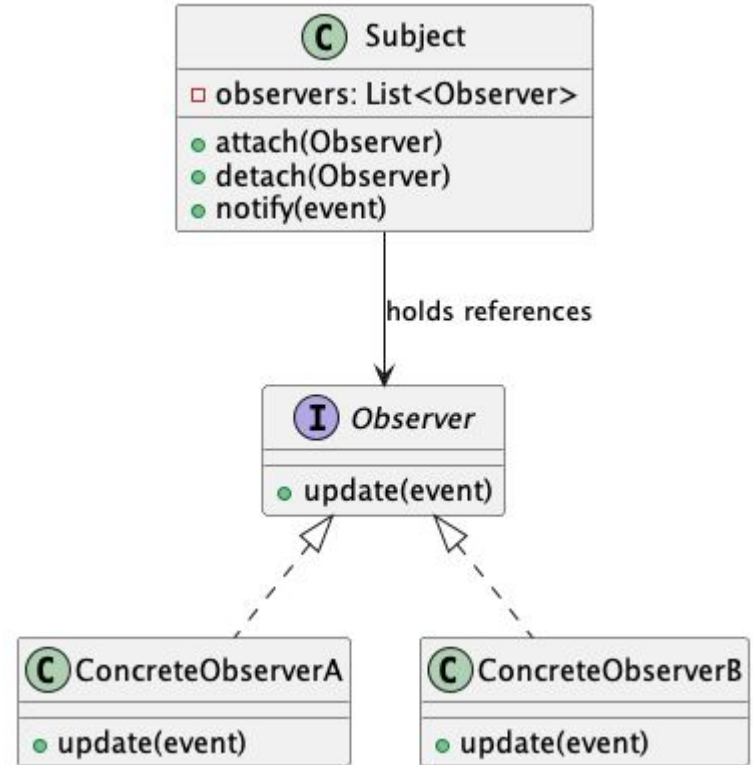
# Factory Method

- Defer object creation to subclasses or dedicated factories.
- Useful when creation logic is complex or varies.



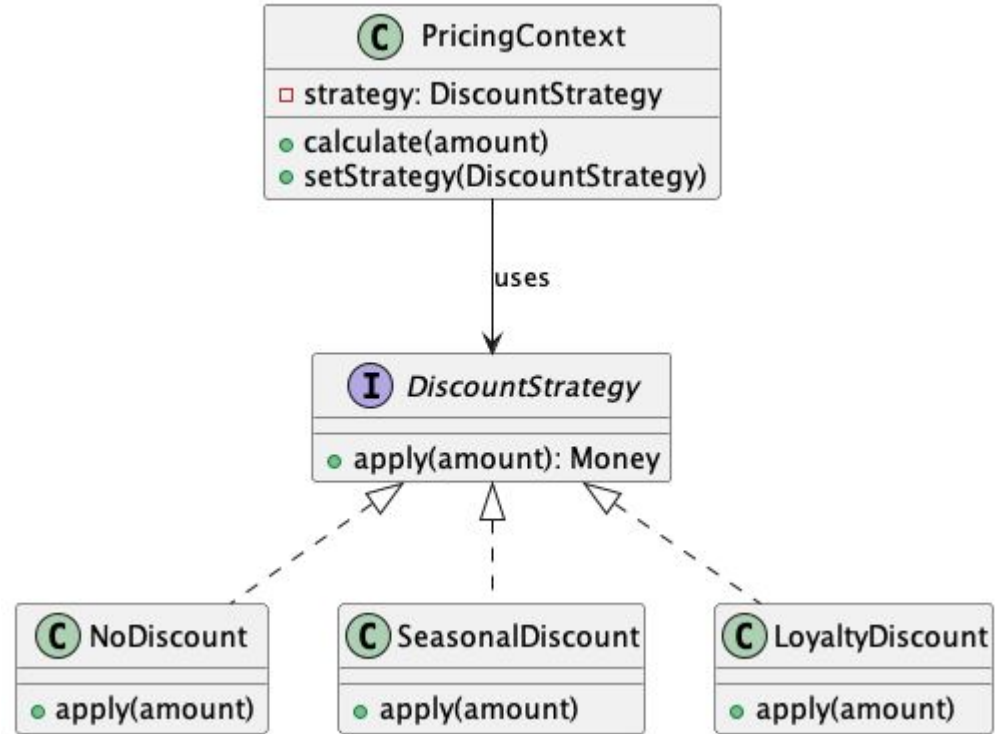
# Observer

- One-to-many update propagation.
- Decouples publishers and subscribers.
- Examples
  - UI events.
  - Event buses.
  - Reactive streams.



# Strategy

- Swap algorithms at runtime without altering the calling code.



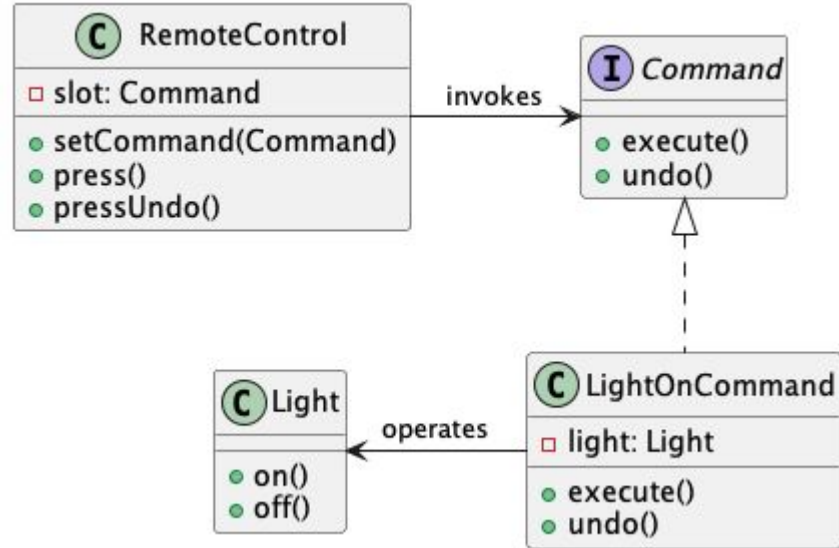
# Command

- Purpose

- Encapsulate requests as objects.
- Enables undo/redo, logging, queueing.

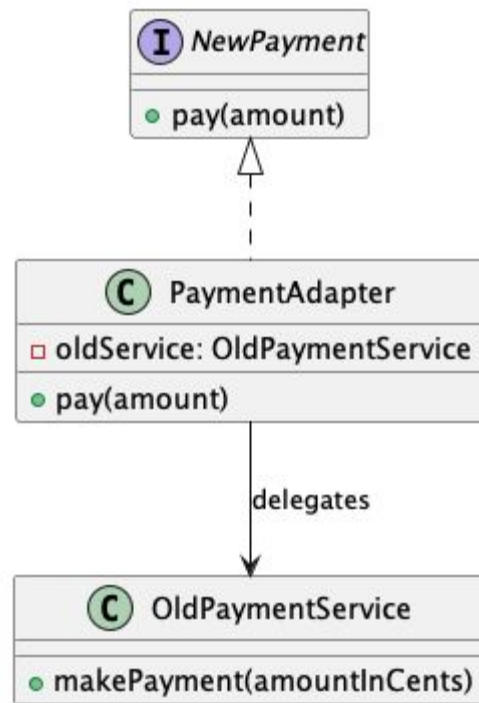
- Real Use

- Task queues.
- GUI action handling.



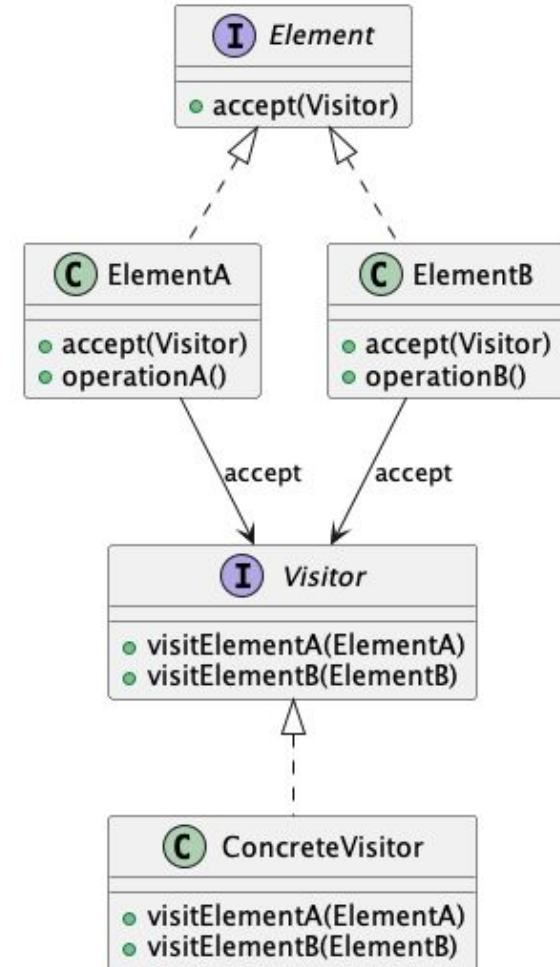
# Adapter

- Allow incompatible interfaces to work together.
- Example
  - Wrapping an old payment provider to match the new provider API.
- Pitfall:
  - Too many adapters may hint at deeper architecture misalignment.



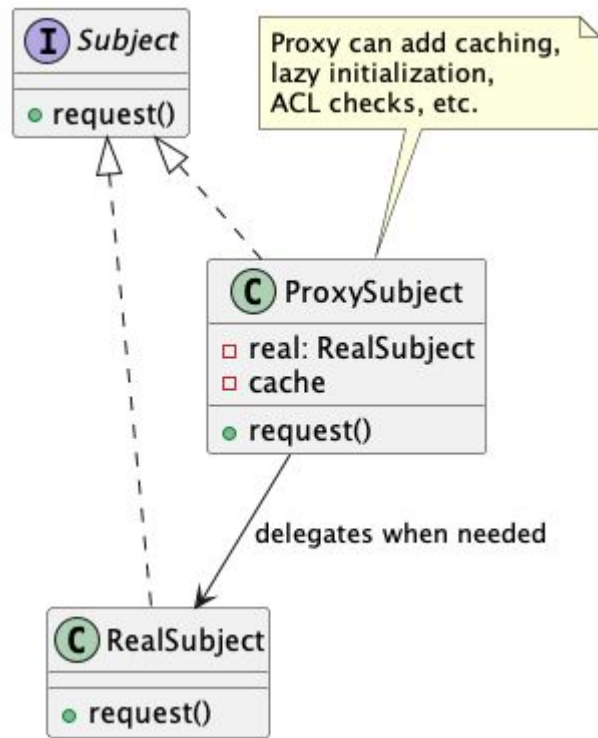
# Visitor

- Add operations to an object structure without modifying the objects.
- Use Cases
  - Compilers (AST visitors).
  - Data export pipelines.
- Requires stable object structure; new operations change frequently.




# Proxy

- Stand-in for a real object with extra behavior like caching, lazy loading, rate-limiting.
- Examples
  - Database lazy loading proxies.
  - Security proxies restricting access.



# Key Takeaways

- 
- Principles guide everyday design decisions.
  - Patterns offer reusable solutions to recurring problems.
  - Good design is evolutionary: refactor continuously as the system grows.
  - Keep code flexible but avoid speculative architecture (balance OCP with YAGNI).
  - Simplicity is often the strongest design principle.