Introduction to Software Design

Interfaces and Modules: Building Better Software

The Most Important SWE Tool

What is Software Design?

> "Software design is about interfaces and modules"

- Just like *architectural blueprints* guide building construction, *software design* guides how we structure and organize our code.

Key Components:

- **Modules**: Self-contained units of functionality
- **Interfaces**: Contracts that define how modules communicate

Real-World Analogy: Building a House

House Building = Software Development

- **Blueprint** → Software Design
- **Rooms** → Modules
- **Doors/Windows** → Interfaces
- **Plumbing/Electrical** → Dependencies

Why Do We Need Software Design?

```python
# Bad: Everything mixed together
def process_user_data():
    email = input("Enter email: ")
    if "@" not in email:
        print("Invalid email")
        return
    password = input("Enter password: ")
    if len(password) < 8:
        print("Password too short")
        return
    # Database connection mixed with validation
    conn = sqlite3.connect("users.db")
    # ... more mixed responsibilities
```

With Good Design (Modular)

```python
# Good: Separated concerns
class EmailValidator:
    def is_valid(self, email: str) -> bool:
        return "@" in email and "." in email

class PasswordValidator:
    def is_valid(self, password: str) -> bool:
        return len(password) >= 8

class UserService:
    def __init__(self, email_validator,
                 password_validator):
        self.email_validator = email_validator
        self.password_validator = password_validator
```

The Problems Without Design

Like a house without blueprints:

- **Maintenance Nightmare**: Hard to fix bugs
- **Difficult to Extend**: Adding features breaks existing code
- **Team Confusion**: Developers can't understand each other's code
- **Testing Issues**: Can't test individual parts
- **Reusability**: Can't reuse components in other projects

What Are Modules?

Module = A Room in Your House

Each room has:

- **Single Purpose** (bedroom for sleeping, kitchen for cooking)
- **Clear Boundaries** (walls separate rooms)
- **Internal Organization** (furniture arrangement)

```python
# Module: User Authentication
class AuthenticationModule:
    def login(self, username, password):
        # Handle login logic
        pass

    def logout(self, user_session):
        # Handle logout logic
        pass
```

What Are Interfaces?

Interface = Doors and Windows

Interfaces define:

- **How to enter/exit** (method signatures)
- **What you can expect** (return types)
- **Rules of interaction** (contracts)

- Interfaces have only the API name and parameters, no code.

```python
from abc import ABC, abstractmethod

class PaymentProcessor(ABC):  # Interface
    @abstractmethod
    def process_payment(self, amount: float) -> bool:
        pass

    @abstractmethod
    def refund_payment(self, transaction_id: str) -> bool:
        pass
```

Interface Implementation Example

- Interfaces should be implemented to be used.

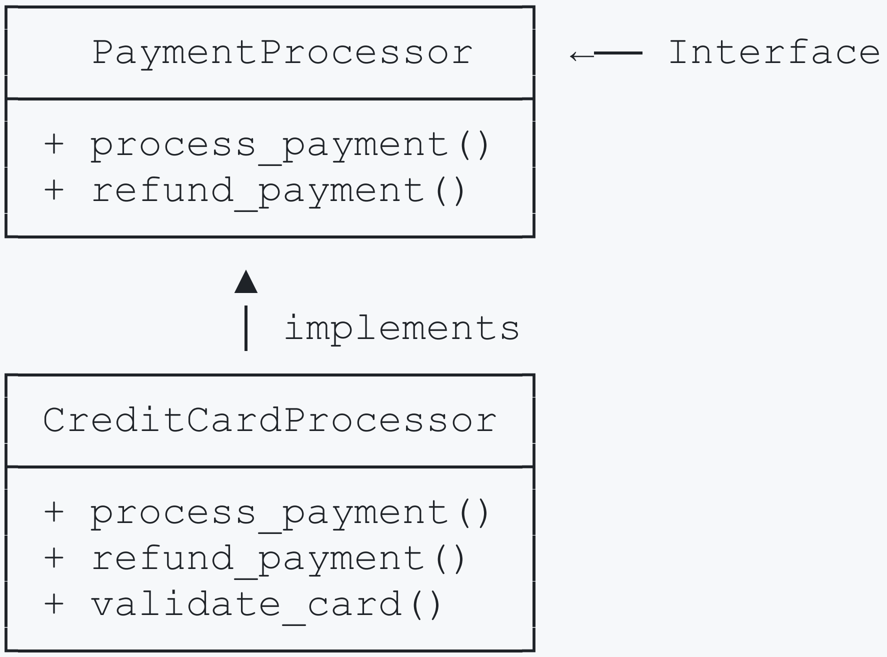
```python
class CreditCardProcessor(PaymentProcessor):
    def process_payment(self, amount: float) -> bool:
        # Credit card specific logic
        print(f"Processing ${amount} via credit card")
        return True

    def refund_payment(self, transaction_id: str) -> bool:
        # Credit card refund logic
        return True
```

Design Tools: UML (Unified Modeling Language)

UML = Architectural Drawings for Software

Class Diagram Example



PaymentProcessor

← Interface

+ process_payment()
+ refund_payment()

implements

CreditCardProcessor

+ process_payment()
+ refund_payment()
+ validate_card()

Design Patterns: Proven Solutions

- **Templates** for solving common programming problems
- **Not finished code** - but proven approaches
- **Reusable solutions** that save time and effort

Why Use Design Patterns?

- **Proven solutions** - tested by thousands of developers
- **Common vocabulary** - easier team communication
- **Maintainable code** - cleaner, more organized
- **Avoid reinventing the wheel**

Real World Issues and Solutions

- **Problem 1:** Creating objects is too complicated
- **Problem 2:** Need different ways to do the same task
- **Problem 3:** Many objects need to know when something changes

Problem 1: Object Creation is Complex

Car Factory Example

- **Imagine ordering a car:**
  - "I want a luxury car."
  - Factory figures out: Sedan + V6 engine + leather seats + premium sound

- **Without pattern:** You tell them every single detail
- **With Factory pattern:** say "luxury" and factory handles the rest!

Problem 2: Different Ways to Do Same Task

Shipping Calculator Example

- **Imagine shipping a package at USPS:**
    - "I want to ship my package."
    - USPS asks What do you want: slow + cheap vs fast + expensive

- **Without Strategy pattern:** Same method for all different cases
- **With Strategy pattern:** Easily switch between methods!

Problem 3: Many Objects Need Updates

Instagram Notification Example

- **Imagine following a celebrity on Instagram:**
  - A celebrity posts a photo
  - 1000 followers should get notified
  - A celebrity can't call each follower individually!

- **Without Observer pattern:** The Celebrity should call each follower individually!
- **With Observer pattern:** Followers "watch" celebrity, get auto-notified!

Pattern Solutions

Problem	Example	Pattern	How it Helps
Complex object creation	Car factory	**Factory**	Just say what you want
Different algorithms	Shipping options	**Strategy**	Easy to switch methods
Many objects need updates	Social media	**Observer**	Auto-notify everyone

Design Patterns in Action

Problem	Pattern Example	Benefit
Object creation complexity	Factory Method	Simplifies object creation logic
Varying algorithms	Strategy	Allows easy swapping of algorithms
Repeated object interaction	Observer	Decouples objects for better flexibility

Refactoring

- Continuous Improvement Through Code Renovation
  - The improvement starts from software design (modules and interfaces)
  - Testing should be followed after refactoring

What is Refactoring?

Refactoring = Home Renovation

- **Same house, better structure**
- **Improve without changing what it does**
- **Make it easier to live in (maintain)**
- **Increase value over time**

Goal: Better code structure without changing functionality

Why Refactor?

Before Renovation:

- Hard to find things
- Difficult to fix problems
- No room for new features
- Scary to make changes

After Renovation:

- Clean and organized
- Easy to maintain
- Space for improvements
- Pleasant to work with

Types of Refactoring = Room Renovations

Refactoring Type	Home Analogy	Code Example
Extract Method	Separate kitchen from living room	Split big functions
Rename Variables	Label storage boxes clearly	`x` → `totalPrice`
Remove Duplicates	One tool shed, not three	Shared utility functions
Simplify Conditions	Clear hallway paths	Reduce nested if-statements

Code Smells: Warning Signs

Code Smells = Signs Your House Needs Repair

Common Code Smells

1. **Long Method** (Room too crowded)

```python
def process_order():  # 200 lines of code!
    # Validation, calculation, database ...
```

2. **Large Class** (Room doing too many things)

```python
class User:  # 50 methods!
    def login(self): pass
    def calculate_taxes(self): pass
    def send_email(self): pass
    def generate_report(self): pass
```

3. **Duplicate Code** (Same room built multiple times)

```
# In multiple places:
if user.age >= 18 and
   user.has_license and
   user.passed_test:
    # Allow driving
```

4. **Feature Envy** (Room accessing neighbor's stuff)

```
class Order:
  def calculate_shipping(self):
    # Accessing too much of the customer's data
    return self.customer.address.state.tax_rate * 0.1
```

5. **God Object** (One room controls the entire house)

```
class SystemManager:  # Controls everything!
    def manage_users(self): pass
    def handle_payments(self): pass
    def generate_reports(self): pass
```

Benefits of Good Software Design

Like a well-designed house:

- **Maintainable**: Easy to fix and update
- **Extensible**: Easy to add new features
- **Testable**: Can test each room independently
- **Reusable**: Rooms can be used in other houses
- **Understandable**: New residents can navigate easily
- **Reliable**: Strong foundation prevents collapse