Software Design Principles
软件设计原则

Fahad Sabah

TFSE_Fall2025_Beijing University of Technology, China
TFSE_Fall2025_北京工业大学，中国

CONTENTS

Planning
Requirements
Design
Implementation
Testing
Deployment
Maintenance

1: The Foundation-Core Design Principles
1：基础-核心设计原则

- ❑Modularity: Breaking systems into manageable pieces
- ❑Abstraction: Hiding complexity behind simple interfaces
- ❑Encapsulation: Protecting data and controlling access

- ❑Library Management System Example

The Problem with Messy Code 凌乱代码的问题

- What does messy library code look like?

```
# Everything in one giant script - THE "BIG BALL OF MUD"
books = [ ]
members = [ ]

def handle_book_return(book_isbn, member_id):
  # Spaghetti code
    for book in books:
        if book["isbn"] == book_isbn:
            for member in members:
                if member["id"] == member_id:
                    if book_isbn in member["borrowed_books"]:
                        book["available"] = True
                        member["borrowed_books"].remove(book_isbn)
                        # And 50 more lines of mixed logic...
```

Problems with this approach:

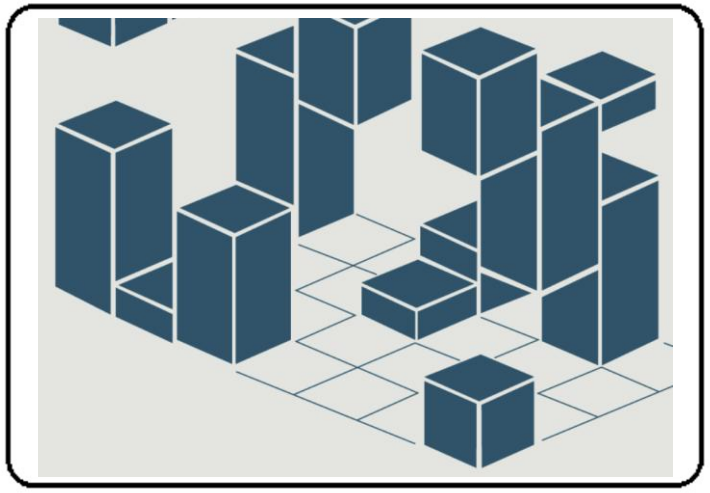
- ❌ Hard to find and fix bugs
- ❌ Difficult to test individual parts
- ❌ Impossible for multiple people to work on
- ❌ Changing one thing breaks everything else

这种方法的问题：

- ❌ 难以发现和修复错误
- ❌ 难以测试单个部件
- ❌ 多人无法处理
- ❌ 改变一件事会破坏其他一切

Principle 1: Modularity 原则一：模块化

- Definition: Breaking a system into smaller, focused modules
- Library Analogy:
  - Fiction section ≠ Science section ≠ Administration office
  - Each has specific responsibilities
  - Clear boundaries between them
- In Code: Separate classes for different responsibilities
  - # INSTEAD OF one giant script...
  - # WE CREATE focused modules



Modularity - Bad Example (What to Avoid)
模块化 - 坏例子（要避免什么）

```
# Hard to understand and change
def library_system():
    books = [{"title": "Book1", "available": True}]
    members = [{"name": "Alice", "books": []}]

    # Mixed up logic
    book = books[0]
    member = members[0]

    if book["available"] and len(member["books"]) < 3:
        book["available"] = False
        member["books"].append(book["title"])
        print("Book borrowed")
    # ... and 100 more lines mixed together
```

Why this is bad?

- ❑ One change can break unrelated functionality
- ❑ Hard to understand what the code does
- ❑ Difficult to test specific features

为什么这不好?

- ❑ 一个更改可能会破坏不相关的功能
- ❑ 难以理解代码的作用
- ❑ 难以测试特定功能

One Change Can Break Unrelated Functionality
一项更改可能会破坏不相关的功能

```
# Example: We want to change max books from 3 to 5
def library_system():
    books = [{"title": "Book1", "available": True}]
    members = [{"name": "Alice", "books": []}]

    # Change this line (easy, right?)
    if book["available"] and len(member["books"]) < 5:  # Changed 3 to 5

    # But wait! 50 lines later...
    if len(member["books"]) > 3:  # Oops! Forgot to change this one!
        print("Member has too many books")

    # And another 30 lines later...
    if len(member["books"]) >= 3:  # And this one!
        charge_overdue_fee()

 # One simple change broke multiple parts!
```

Everything is mixed together, so changing one part can accidentally break another.
所有东西都混合在一起，因此更换一个部分可能会意外损坏另一个部分。

What happens:

- ❑ You change the rule in one place
- ❑ But miss it in other places
- ❑ Now the system has inconsistent rules
- ❑ Some parts allow 5 books, others only allow 3

发生什么：

- ❑ 您可以在一个地方更改规则
- ❑ 但在其他地方却错过了
- ❑ 现在系统规则不一致
- ❑ 有些部分允许 5 本书，有些部分只允许 3 本书

Hard to Understand What the Code Does

很难理解代码的作用

```
def library_system():
    # Line 1: Set up books
    books = [{"title": "Book1", "available": True}]
    # Line 2: Set up members
    members = [{"name": "Alice", "books": []}]
    # Line 3: Some random calculation?
    x = 5 + 3
    # Line 4: Borrowing logic (but mixed with other stuff)
    if book["available"] and len(member["books"]) < 3:
        book["available"] = False
    # Line 5: Print something?
    print("Processing...")
    # Line 6: More borrowing logic, but now with email?
    if not book["available"]:
        send_email("Book unavailable")
    # Line 7: Database save mixed with borrowing logic
    save_to_database(book)
    # ... and 100 more mixed-up lines
```

It's like reading a book where all the sentences are mixed up.
这就像读一本书，所有的句子都混淆了。

Try to answer these simple questions:
尝试回答这些简单的问题：

- ❑ Where does the borrowing process start and end?
- ❑ 借贷过程从哪里开始和结束？
- ❑ What are all the steps involved in borrowing a book?
- ❑ 借书涉及哪些步骤？
- ❑ Where would I add a new step (like sending an SMS notification)?
- ❑ 我在哪里添加新步骤（例如发送短信通知）？
- ❑ You'd have to read all 100+ lines to understand!
- ❑ 您必须阅读所有 100+ 行才能理解！

Difficult to Test Specific Features
难以测试特定功能

 # How would you test JUST the borrowing logic?

```
def test_borrowing():
    # Can't test borrowing alone - it's mixed with everything!
```

Problems with testing the messy code [测试凌乱代码的问题]

- ❑ 1. Need to set up entire system (books, members, database, email)
- ❑ 2. Can't test "borrowing" without testing email sending, database saving, etc.
- ❑ 3. If email system is down, borrowing test fails (even though borrowing works!)
- ❑ 4. Hard to test edge cases (what if book is already borrowed? what if member has max books?)

Simple Before/After Examples 简单的前后示例

```
def do_everything():
    # Mixed responsibilities
    book = get_book()
    member = get_member()

    # Borrowing logic
    if book.available and len(member.books) < 3:
        book.available = False
        member.books.append(book)

    # Email logic mixed in
    send_email("Book borrowed")

    # Database logic mixed in
    save_to_database(book)
    save_to_database(member)

    # Reporting mixed in
    generate_report()

    # 100 more mixed responsibilities...
```

```
# Separate functions - each does ONE thing
def borrow_book(book, member):
    if can_borrow(book, member):
        book.borrow()
        member.add_book(book)
        return True
    return False

def can_borrow(book, member):
    return book.is_available and member.can_borrow_more()

# Usage:
def process_borrowing(book_id, member_id):
    book = find_book(book_id)
    member = find_member(member_id)

    if borrow_book(book, member):
        notify_borrowing(book, member)      # Separate responsibility
        save_borrowing_record(book, member) # Separate responsibility
        return True
    return False
```

Inputs

Result?

Real Consequences of Bad Design Scenario
糟糕设计场景的实际后果

- Library wants to add a new rule - "Students can borrow 5 books, staff can borrow 10“
- 图书馆想增加一条新规则——“学生可以借5本书，教职员工可以借10本书”

- With messy code:
  - ❑Have to find and change the rule in multiple places
  - ❑Risk of missing some places
  - ❑Takes hours to make a simple change
  - ❑High chance of breaking something

With clean code 使用干净的代码

```
# Change one simple function
def can_borrow_more(member):
    if member.type == "student":
        return len(member.books) < 5
    elif member.type == "staff":
        return len(member.books) < 10
    else:
        return len(member.books) < 3
# Done in 5 minutes, no risk of breaking other features
```

Modularity-Good Example 模块化 - 很好的例子

 # SEPARATE CLASSES for separate concerns

```
class Book:
    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self.isbn = isbn
        self.is_available = True

class Member:
    def __init__(self, name, member_id):
        self.name = name
        self.member_id = member_id
        self.borrowed_books = []

class Library:
    def __init__(self):
        self.books = []
        self.members = []
```

 # Each class does one thing well

Benefits of this approach:

- ☑ Easy to understand each class's purpose
- ☑ Can work on Book class without touching Member class
- ☑ Easy to test each class independently
- ☑ Clear boundaries and responsibilities

这种方法的好处：

- ☑ 易于理解每个课程的目的
- ☑ 可以在不接触 Member 类的情况下处理 Book 类
- ☑ 易于独立测试每个类别
- ☑ 明确的界限和责任

Principle 2: Abstraction 原则 2：抽象

- Definition: Hiding complex implementation details behind simple interfaces
- Library Analogy:
  - Members/Patrons use a simple catalog system
  - They don't need to know about database queries, indexing, or storage
  - Complexity is hidden behind a simple interface
- In Code: Using abstract classes and interfaces to simplify usage

Abstraction Example - Notification System
抽象示例 - 通知系统

```
# COMPLICATED WAY
def send_email(to, subject, message):
    # 50 lines of email server code...
    print(f"Email sent to {to}")

# SIMPLE WAY (Abstraction)
def notify_member(message):
    # We don't care HOW email works
    send_email("member@email.com", "Notification", message)

# Usage: Simple and clean
notify_member("Your book is due tomorrow")
```

You drive a car without knowing engine mechanics!
你在不了解发动机机械原理的情况下驾驶汽车!

Abstraction Benefit: We can change from email to SMS without breaking our library system!
抽象优势：我们可以在不破坏图书馆系统的情况下从电子邮件更改为短信!

Principle 3: Encapsulation 原则 3：封装

- Definition: Grouping/Bundling data with methods that operate on that data, and restricting direct access
- Library Analogy:
  - You can't just walk into the library storage and take books
  - You must go through the circulation desk (controlled access)
  - Librarians ensure rules are followed (data validation)
- In Code: Using private attributes and public methods

Encapsulation Example - Protecting Book Data
封装示例 - 保护书籍数据

```
book = {"title": "Python", "available": True}
book["available"] = "hello"  # This makes no sense!

class Book:
    def __init__(self, title):
        self.title = title
        self._available = True  # "Private" variable

    def borrow(self):
        if self._available:
            self._available = False
            return True
        return False  # Can't borrow if not available

    def return_book(self):
        self._available = True
```

BAD: Anyone can change your data

Encapsulation Benefit: Prevents invalid book states and maintains data integrity!

GOOD: Control how data changes

Encapsulation Example-Protecting Library Rules
封装示例 - 保护库规则

```
class Member:
    def __init__(self, name):
        self.name = name
        self._borrowed_books = []  # Protected list

    def borrow_book(self, book):
        if len(self._borrowed_books) < 3:  # Max 3 books rule
            self._borrowed_books.append(book)
            return True
        return False  # Can't borrow more than 3

    def return_book(self, book):
        if book in self._borrowed_books:
            self._borrowed_books.remove(book)
```

Simple Example – After

简单示例 - 之后

```python
# MODULARITY: Separate classes
class Book:
    def __init__(self, title):
        self.title = title
        self._available = True

    # ENCAPSULATION: Control access
    def borrow(self):
        if self._available:
            self._available = False
            return True
        return False
```

1

```python
class Member:
    def __init__(self, name):
        self.name = name
        self._books = []

    def borrow_book(self, book):
        if len(self._books) < 3 and book.borrow():
            self._books.append(book)
            return True
        return False
```

2

```python
# ABSTRACTION: Simple interface
def borrow_book_for_member(book, member):
    return member.borrow_book(book)  # Hide complexity
```

3

Why Bother? 何苦?

```
# BAD DESIGN: Change is hard
# Need to change max books from 3 to 5?
# Search through 1000 lines of code...

# GOOD DESIGN: Saves Time
# GOOD DESIGN: Change is easy
class Member:
    MAX_BOOKS = 3  # Change this one line!

    def borrow_book(self, book):
        if len(self._books) < self.MAX_BOOKS:
            # ...
```

Good design = Less headaches later!
好的设计 = 以后少头疼!

Quick Practice [快速练习]

- **Spot the Principle**

```
class EmailSender:
    def send_welcome(self): pass
class SMSSender:
    def send_reminder(self): pass
```

? ☑ **Modularity** - Separate message senders
☑ 模块化 - 单独的消息发送者

```
def login(username, password):
    # We don't see password checking details
    return check_credentials(username, password)
```

? ☑ **Abstraction** - Hide complex security checks
☑ 抽象 - 隐藏复杂的安全检查

```
class BankAccount:
    def __init__(self):
        self._balance = 0  # Can't access directly!

    def withdraw(self, amount):
        if amount <= self._balance:  # Validation
            self._balance -= amount
```

?

- ☑ **Encapsulation** - Protect money with rules
- ☑ 封装 - 用规则保护资金

```
# Simple usage:
account = BankAccount(100)  # Start with 元100
account.withdraw(30)        # Withdraw 元30
```

Summary [总结]

- Three Simple Rules
  - Modularity: Break big code into small pieces
  - Abstraction: Hide complicated stuff behind simple interfaces
  - Encapsulation: Protect data with controlled access

- 结果：代码更易于阅读、更改和修复！
- Result: Code that's easier to read, change, and fix!

2: The SOLID Foundation – (SRP, OCP, LSP, ISP, DIP)

What is SOLID? [什么是 SOLID？]

- Five Letters = Five Principles [五个字母=五项原则]
  - S - Single Responsibility Principle [S - 单一责任原则]
  - O - Open/Closed Principle [O - 开/闭原理]
  - L - Liskov Substitution Principle [L - 利斯科夫替代原理]
  - I - Interface Segregation Principle [I - 接口分离原理]
  - D - Dependency Inversion Principle [D - 依赖反转原理]
- Simple Goal: Make code easy to change and hard to break

  简单目标：使代码易于更改且难以破坏

S - Single Responsibility Principle [S - 单一责任原则]

- One Job Per Class
- Rule: "A class should have only one reason to change"
- Simple Translation: Each class should do exactly one thing

THINK: Specialized workers vs. doing everything yourself
[思考：专业工人与自己做所有事情]

SRP - Bad Example The "Do-It-All" Class
SRP - 坏例子 “Do-It-All”类

```python
class LibraryManager:
    def add_book(self, book): pass       # Job 1: Book management
    def remove_book(self, book): pass    # Job 1: Book management

    def register_member(self, member): pass  # Job 2: Member management
    def remove_member(self, member): pass    # Job 2: Member management

    def send_overdue_emails(self): pass      # Job 3: Notifications
    def calculate_fines(self): pass          # Job 4: Finance
    def generate_reports(self): pass         # Job 5: Reporting
    def backup_database(self): pass          # Job 6: Database
```

Problem: Too many reasons to change = fragile code!

问题：改变的理由太多 = 代码脆弱！

SRP - Good Example [Specialized Classes]

SRP - 好例子 [专业课程]

```python
class BookManager:
    def add_book(self, book): pass
    def remove_book(self, book): pass

class MemberManager:
    def register_member(self, member): pass
    def remove_member(self, member): pass

class NotificationService:
    def send_overdue_emails(self): pass

class FinanceService:
    def calculate_fines(self): pass

class ReportGenerator:
    def generate_reports(self): pass
```

Benefit: Change one thing without breaking others!

好处：改变一件事而不破坏其他事情!

SRP - Simple Library Example [Before vs. After]
SRP - 简单库示例 [之前与之后]

```python
# BEFORE: Mixed responsibilities
class Library:
    def borrow_book(self): pass      # Book logic
    def save_to_database(self): pass # Database logic
    def print_receipt(self): pass    # Printing logic

# AFTER: Separate responsibilities
class Library:
    def borrow_book(self): pass      # Only book logic

class DatabaseService:
    def save_borrowing(self): pass   # Only database logic

class PrinterService:
    def print_receipt(self): pass    # Only printing logic
```

Easier to test, change, and understand!
更易于测试、更改和理解!

O - Open/Closed Principle [O - 开/闭原理]

- Open for Extension, Closed for Modification
- Rule: "You should be able to add new features without changing existing code"
- Simple Translation: Add new things easily, don't rewrite old things

OCP - Bad Example [The "If-Else" Monster]
OCP - 坏例子 [“If-Else”怪物]

```python
def calculate_area(shape_type, dimensions):
    if shape_type == "circle":
        radius = dimensions["radius"]
        return 3.14 * radius * radius
    elif shape_type == "rectangle":
        width = dimensions["width"]
        height = dimensions["height"]
        return width * height
    elif shape_type == "triangle":
        base = dimensions["base"]
        height = dimensions["height"]
        return 0.5 * base * height
    # Adding new shape? Modify this function!
```

Problem: Have to change working code to add new features

问题：必须更改工作代码才能添加新功能

OCP - Library Example [Adding New Book Types]
OCP - 图书馆示例 [添加新的书籍类型]

```python
class Book(ABC):
    @abstractmethod
    def get_late_fee(self, days_late):
        pass

class RegularBook(Book):
    def get_late_fee(self, days_late):
        return days_late * 0.50  # 50 cents per day

class ReferenceBook(Book):
    def get_late_fee(self, days_late):
        return days_late * 1.00  # $1 per day
```

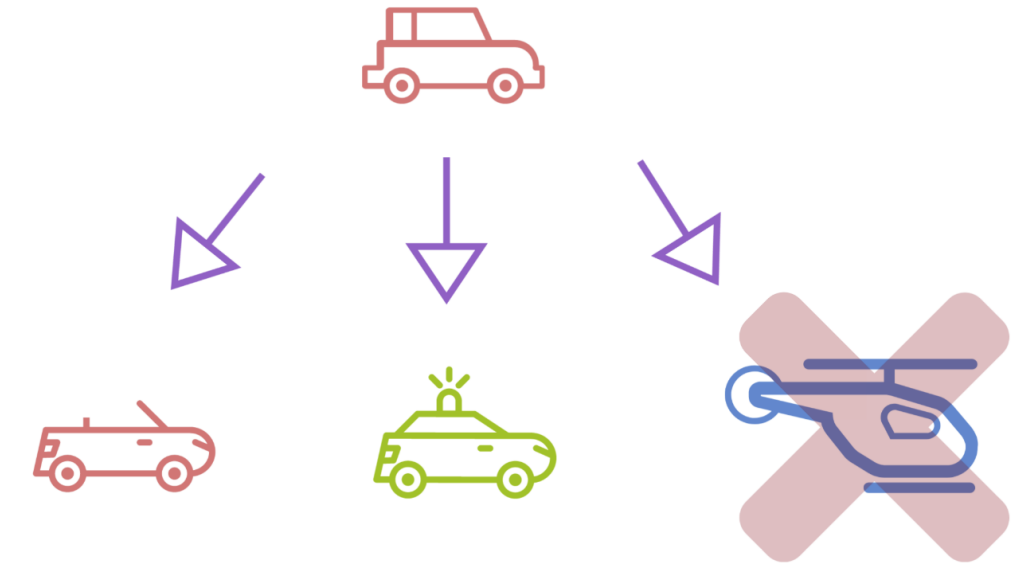
```python
# Easy to add new book types!
class AudioBook(Book):
    def get_late_fee(self, days_late):
        return 0  # No late fees for audio books

class EBook(Book):
    def get_late_fee(self, days_late):
        return days_late * 0.25  # 25 cents per day
```

No need to change existing book classes!
无需更改现有的书籍类别!

L - Liskov Substitution Principle

L - 利斯科夫替代原理

- Substitutes Should Work the Same
- Rule: "You should be able to use a subclass wherever you use its parent class"
- Simple Translation: If it looks like a duck, it should quack like a duck



LSP - The Famous Problem [Square vs. Rectangle]
LSP - 著名问题 [正方形与矩形]

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def set_width(self, width):
        self.width = width

    def set_height(self, height):
        self.height = height

    def area(self):
        return self.width * self.height
```

```python
# Square IS-A Rectangle in geometry...
class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)

    def set_width(self, width):
        self.width = width
        self.height = width  # This breaks the expectation!

    def set_height(self, height):
        self.height = height
        self.width = height  # This breaks the expectation!
```

Problem: Square doesn't behave like a Rectangle should!
问题：正方形的行为不像矩形应该的那样!

LSP - Why It Matters [Unexpected Behavior]
LSP - 为什么它很重要 [意外行为]

```python
def test_rectangle(rect):
    # This should work for any rectangle
    original_area = rect.area()
    rect.set_width(10)      # Only change width
    rect.set_height(20)     # Only change height
    new_area = rect.area()
    return new_area > original_area  # Should be true

# Test with Rectangle
rectangle = Rectangle(5, 5)
print(test_rectangle(rectangle))  # ✅ True - works correctly

# Test with Square (should work the same)
square = Square(5)
print(test_rectangle(square))     # ❌ False - breaks expectations!
```

Square violates LSP: It doesn't behave like a proper Rectangle substitute
Square 违反 LSP：它的行为不像一个合适的 Rectangle 替代品

LSP - Good Example [Proper Inheritance]
LSP - 好例子 [正确继承]

```
class Shape(ABC):
    @abstractmethod
    def area(self): pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side
```

```
# Both can be used as Shapes without surprises!
def print_area(shape):
    print(f"Area: {shape.area()}")  # Works for any Shape

rectangle = Rectangle(4, 5)
square = Square(4)
print_area(rectangle)  # ☑ Area: 20
print_area(square)     # ☑ Area: 16
```

Both behave as expected when treated as Shapes!
当被视为形状时，两者的行为都符合预期!

LSP - Library Example [Proper Book Hierarchy]
LSP - 图书馆示例 [正确的书籍层次结构]

```
class Book(ABC):
    @abstractmethod
    def can_borrow(self):
        pass

class BorrowableBook(Book):
    def __init__(self, available_copies):
        self.available_copies = available_copies

    def can_borrow(self):
        return self.available_copies > 0

class ReferenceBook(Book):
    def can_borrow(self):
        return False  # Reference books cannot be borrowed
```

```
# Both can be used as Books without breaking expectations
def check_availability(book):
    if book.can_borrow():
        print("Available for borrowing")
    else:
        print("Not available for borrowing")

# Works correctly for both types!
regular_book = BorrowableBook(3)
reference_book = ReferenceBook()

check_availability(regular_book)   # ☑ "Available for borrowing"
check_availability(reference_book)  # ☑ "Not available for borrowing"
```

No surprises when substituting!
替换时没有惊喜!

I - Interface Segregation Principle
I - 接口分离原理

- Small, Focused Interfaces

- Rule: "Many client-specific interfaces are better than one general-purpose interface"

- Simple Translation: Don't force classes to implement methods they don't need

ISP - Bad Example [The "Kitchen Sink" Interface]
ISP - 坏例子 [“厨房水槽”接口]

```
class SimplePrinter(LibraryMachine):
    # I'm forced to implement ALL methods, even though I'm just a printer!
    def scan_book(self, book):
        raise NotImplementedError("I'm just a printer!")  # ❌ Doesn't make sense

    def print_receipt(self, receipt):
        print(f"Printing: {receipt}")  # ☑ This is my real job

    def send_email(self, message):
        raise NotImplementedError("I can't send emails!")  # ❌ Doesn't make sense

    def backup_data(self):
        raise NotImplementedError("I don't do backups!")  # ❌ Doesn't make sense

    def calculate_fine(self, days):
        raise NotImplementedError("I'm not an accountant!")  # ❌ Doesn't make sense
```

Problem: A simple printer must implement email, backup, and fine calculations!
问题：一台简单的打印机必须实现电子邮件、备份和精细计算！

What's wrong with this?

1. **Wasted Code**: 4 out of 5 methods just raise errors
2. **Confusing**: Other developers think the printer can do everything
3. **Fragile**: If we change the interface, we break the printer
4. **Violates SRP**: One class has multiple unrelated responsibilities

ISP - Good Example [Focused Interfaces]
ISP - 好例子 [聚焦接口]

```
# GOOD: Separate, focused interfaces
class Scanner(ABC):
    @abstractmethod
    def scan_book(self, book): pass  # Just scanning

class Printer(ABC):
    @abstractmethod
    def print_receipt(self, receipt): pass  # Just printing

class EmailSender(ABC):
    @abstractmethod
    def send_email(self, message): pass  # Just emails

class BackupService(ABC):
    @abstractmethod
    def backup_data(self): pass  # Just backups

class FineCalculator(ABC):
    @abstractmethod
    def calculate_fine(self, days): pass  # Just fines

# Library hires specialized people/equipment
scanner = BookScanner()       # Only scans books ☑
printer = ReceiptPrinter()    # Only prints receipts ☑
email_system = EmailService()  # Only sends emails ☑
accountant = Accountant()     # Only calculates fines ☑
it_guy = ITSpecialist()       # Only does backups ☑
janitor = Janitor()           # Only waters plants ☑

# Each does one job well! ☑
```

Benefit: Classes only implement what they actually need!
好处：类只实现它们实际需要的内容!

D - Dependency Inversion Principle
D - 依赖反转原理

- Depend on Abstractions

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

- Simple Translation: Depend on interfaces (what things do), not specific implementations (how they do it)

Dependency Inversion (Example)
依赖关系反转（示例）

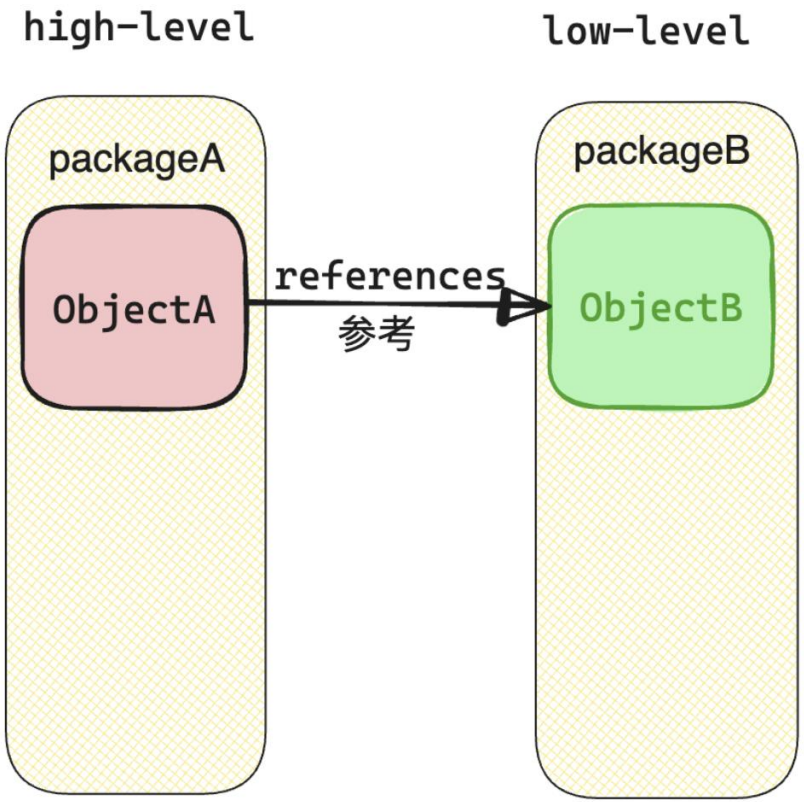


图1

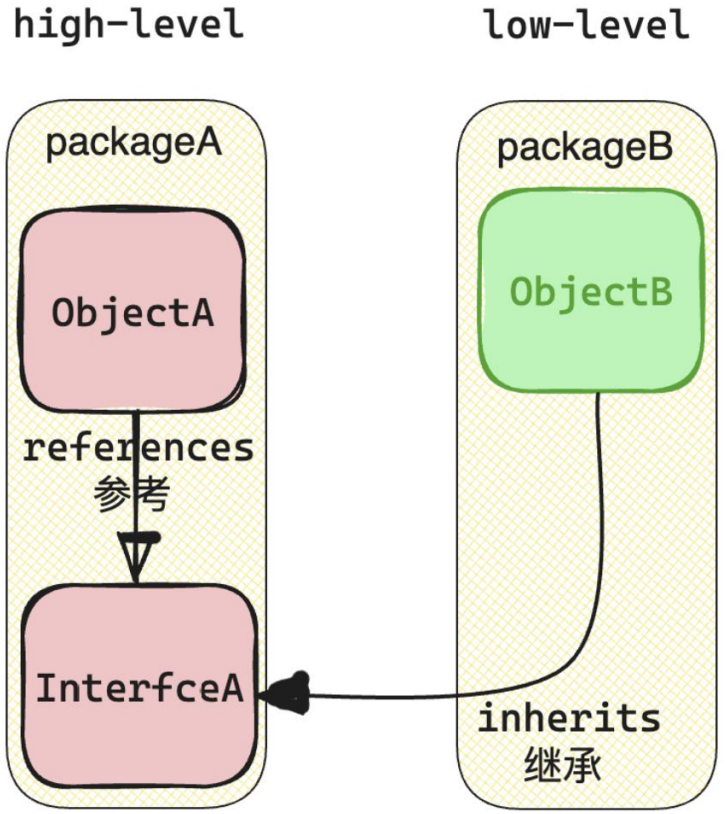
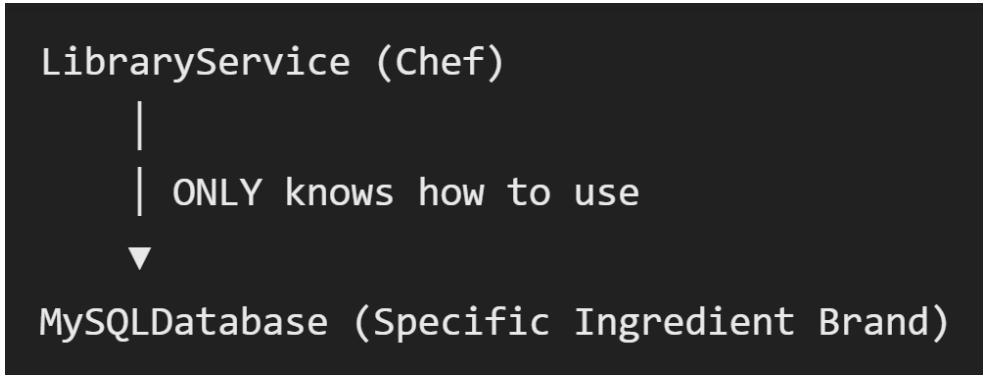
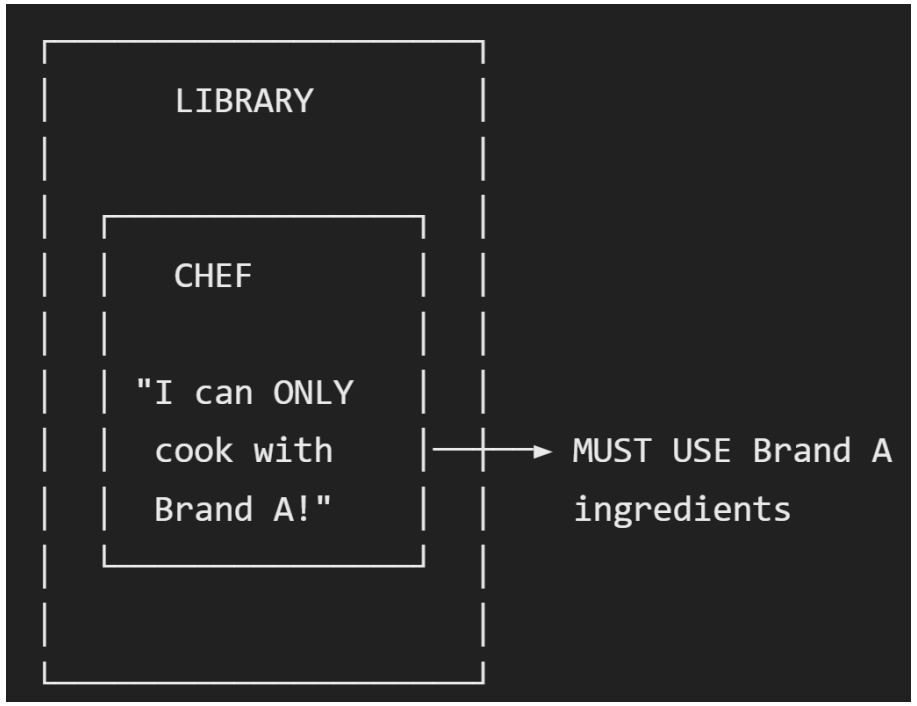


图2

DIP - Bad Example [Tight Coupling]
DIP - 不良示例 [紧耦合]

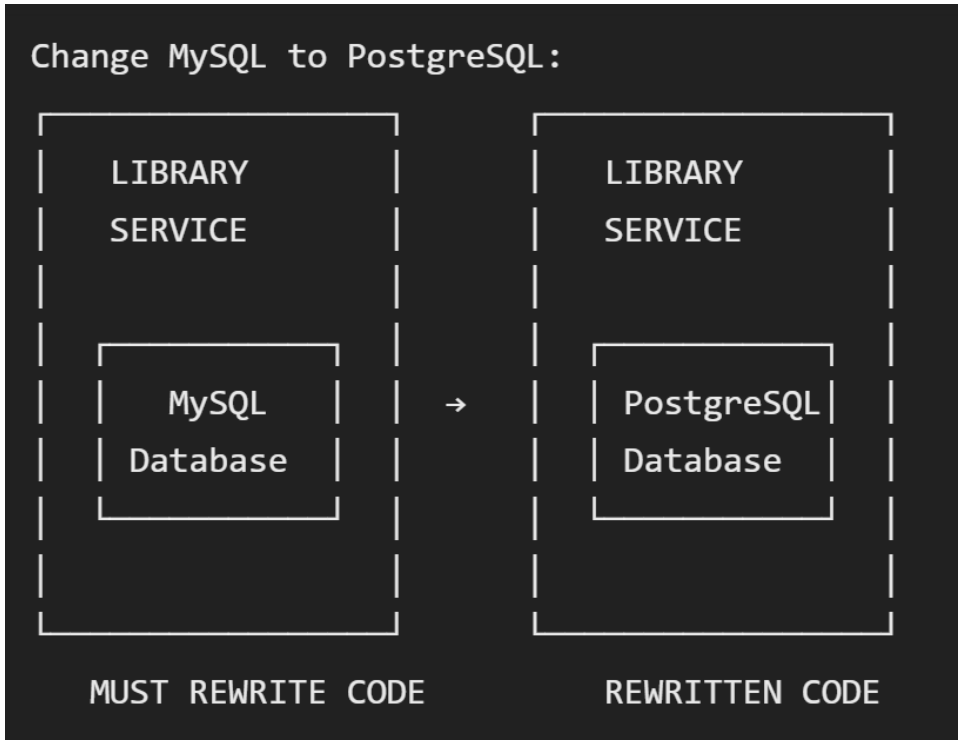




Problem: If Brand A ingredients are unavailable, the chef can't cook!
问题：如果 A 品牌食材不可用，厨师就无法烹饪!

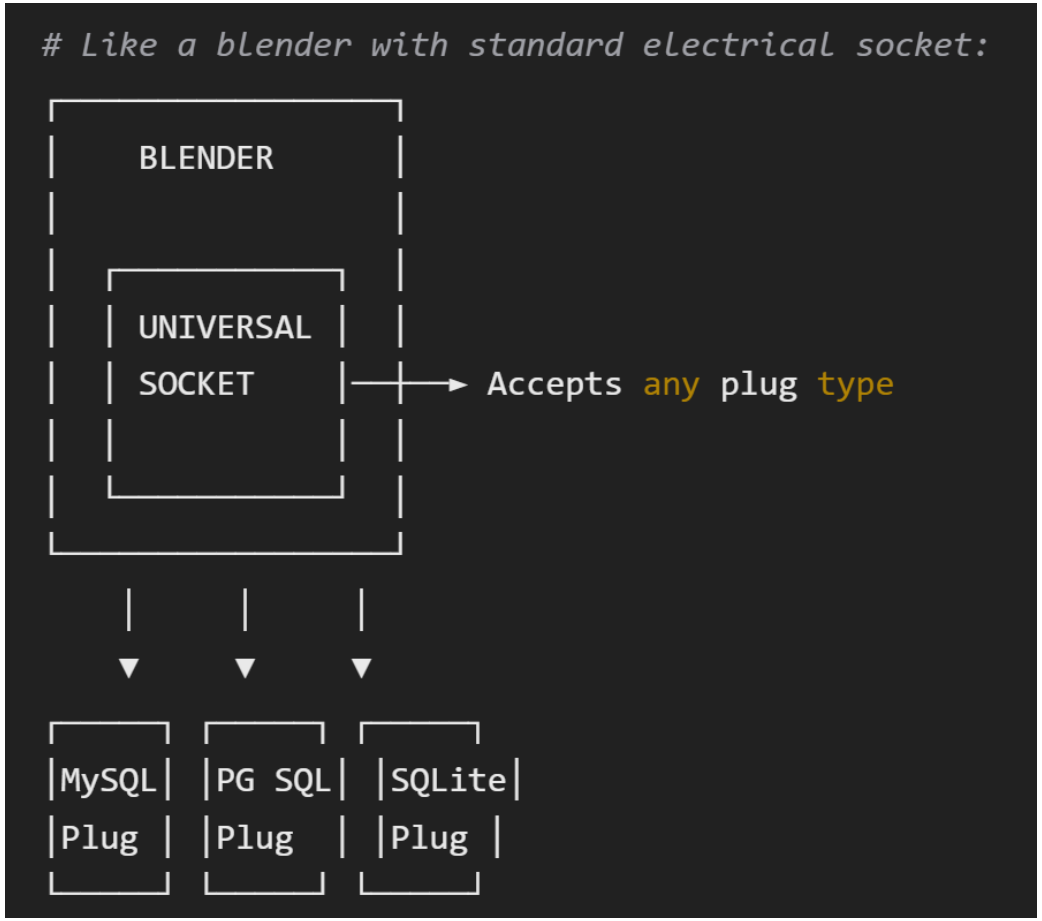
DIP - Bad Example [Tight Coupling]
DIP - 不良示例 [紧耦合]

```
# High-level module depends on low-level module directly
class LibraryService:
    def __init__(self):
        self.database = MySQLDatabase()  # Direct dependency!

    def save_book(self, book):
        self.database.save(book)  # Tightly coupled to MySQL

class MySQLDatabase:
    def save(self, data):
        # MySQL-specific code
        print("Saving to MySQL database")

# Problem: What if we want to switch to PostgreSQL?
# We have to change LibraryService class!
```



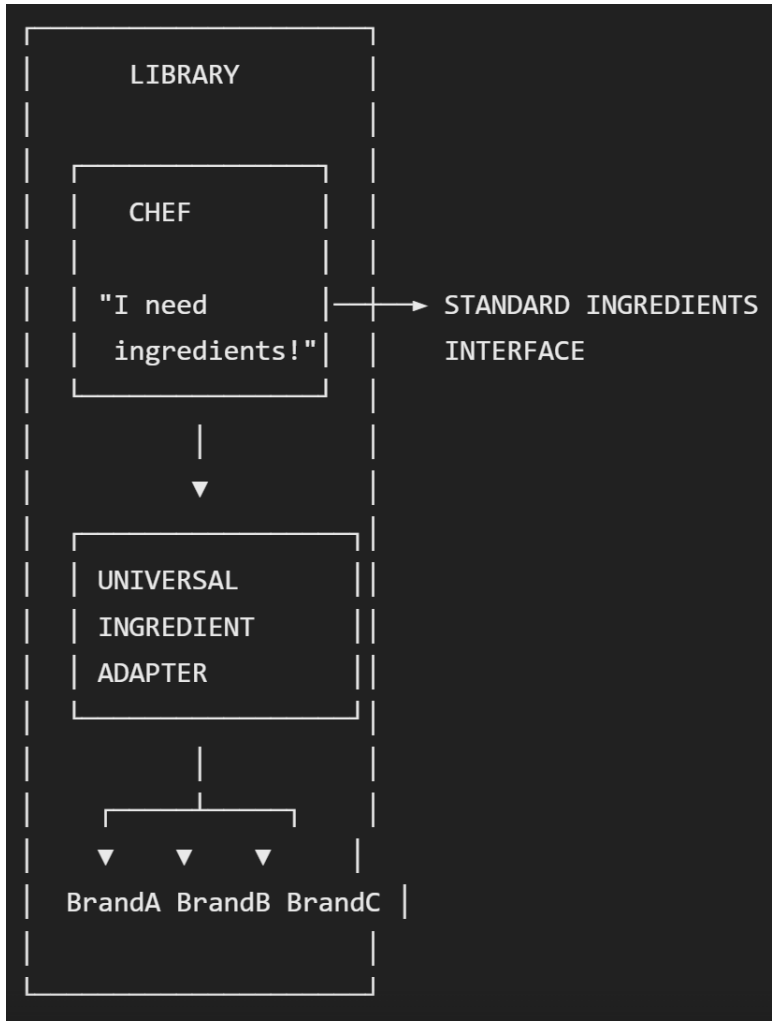
Changing database = Changing business logic!
改变数据库=改变业务逻辑!

GOOD DESIGN: Dependency Inversion (Loose Coupling)
好设计：依赖反转（松耦合）



GOOD DESIGN: Dependency Inversion (Loose Coupling)
好设计：依赖反转（松耦合）

```
LibraryService (Chef)
    │
    │ Requires: "Something that can save data"
    ▼
Database (Interface/Abstraction)
    ▲
    │
    ├── MySQLDatabase (Brand A)
    ├── PostgreSQLDatabase (Brand B)
    └── SQLiteDatabase (Brand C)
```



DIP - Good Example Loose Coupling with Abstractions [DIP - 很好的例子与抽象的松耦合]

```
from abc import ABC, abstractmethod

# Step 1: Create abstraction
class Database(ABC):
    @abstractmethod
    def save(self, data): pass

# Step 2: High-level depends on abstraction
class LibraryService:
    def __init__(self, database: Database):  # Depends on interface!
        self.database = database

    def save_book(self, book):
        self.database.save(book)  # Works with ANY database
```

```
# Step 3: Low-level modules implement abstraction
class MySQLDatabase(Database):
    def save(self, data):
        print("Saving to MySQL database")

class PostgreSQLDatabase(Database):
    def save(self, data):
        print("Saving to PostgreSQL database")

class FileSystemDatabase(Database):
    def save(self, data):
        print("Saving to file system")
```

Benefit: Easy to swap databases without changing LibraryService! 好处：无需更改 LibraryService 即可轻松交换数据库!

DIP - Dependency Injection Passing Dependencies from Outside [DIP - 依赖注入从外部传递依赖项]

```
# Dependencies are "injected" rather than created inside
class LibraryService:
    def __init__(self, database: Database, notifier: Notifier):
        self.database = database
        self.notifier = notifier

    def borrow_book(self, book, member):
        self.database.save_borrowing(book, member)
        self.notifier.send_confirmation(member, book)
```

```
# Usage: We control the dependencies from outside
mysql_db = MySQLDatabase()
email_notifier = EmailNotifier()

library = LibraryService(mysql_db, email_notifier)

# Easy to switch to different implementations
postgresql_db = PostgreSQLDatabase()
sms_notifier = SMSNotifier()

library2 = LibraryService(postgresql_db, sms_notifier)
```

This makes testing much easier!
这使得测试变得更加容易！

Complete SOLID Recap [完整的 SOLID 回]

- All Five Principles Together
  - S - Single Responsibility: One class, one job
  - O - Open/Closed: Easy to extend, hard to modify
  - L - Liskov Substitution: Substitutes work the same
  - I - Interface Segregation: Small, focused interfaces
  - D - Dependency Inversion: Depend on abstractions

- Result: Flexible, maintainable, testable code!

Key Takeaways [关键外]

- Good design starts with simple principles
- 好的设计始于简单的原则
- SOLID principles guide you to better decisions
- SOLID 原则指导您做出更好的决策
- Good design makes change easier and safer
- 好的设计让变更更容易、更安全
- You're now equipped to write professional-quality code!
- 您现在已经准备好编写专业品质的代码了！