

SOLID, DRY, KISS Principles

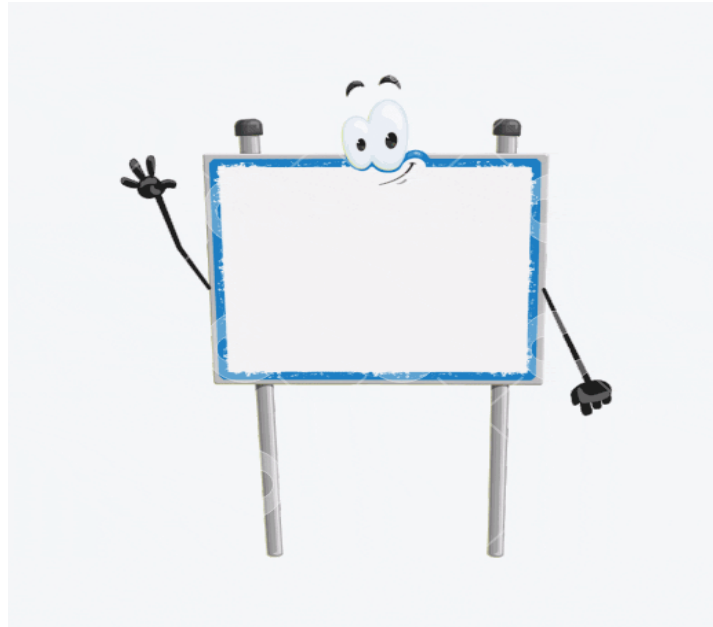
Personal Banking Management System
Software Design Principles Training

Training Objectives

- ✓ Master SOLID principles through practical examples
- ✓ Apply DRY (Don't Repeat Yourself) effectively
- ✓ Implement KISS (Keep It Simple, Stupid) methodology
- ✓ Coding Exercise

Python Class Design Best Practices

Let's inspect whether the class I formed is having issues or not.
Go to file **class_design.py**



Design Flaws

1

Public Attributes

No encapsulation - attributes can be modified freely, allowing invalid data like negative salaries.

2

Poor Naming

Abbreviated names like `sal` reduce code readability and maintainability.

3

No Constructor

Missing `__init__` method allows objects to exist in invalid states.

4

Magic Numbers

Hardcoded values like `1.05` make code rigid and unclear.

5

Mixed Responsibilities

Class handles both data management and presentation, violating Single Responsibility Principle.



Solutions



Encapsulation with Validation

Use private attributes with getter/setter methods and validation logic



Clear Naming Conventions

Replace abbreviations with descriptive names like `_salary`



Robust Constructor

Implement `__init__` with parameter validation for object integrity



Parameterized Methods

Replace magic numbers with parameters for flexibility and clarity



Separation of Concerns

Use `__str__` method for data representation, let caller handle display



Key Takeaways & Best Practices

▶ Always validate input data

▶ Use descriptive variable names

▶ Implement proper constructors

▶ Avoid hardcoded magic numbers

▶ Follow Single Responsibility Principle

▶ Encapsulate data with private attributes

Why is learning design principles important?



Common Problems Without Good Design Principles

- Tightly coupled code that's hard to modify
- Duplicated logic across different modules
- Complex, hard-to-understand code
- Difficulty adding new account types or features
- Testing becomes nearly impossible

SOLID Principles Overview

S - Single Responsibility

A class should have only one reason to change

O - Open/Closed

Open for extension, closed for modification

L - Liskov Substitution


Subtypes must be substitutable for their base types

I - Interface Segregation

Many specific interfaces are better than one general purpose interface

D - Dependency Inversion

Depend on abstractions, not concretions

 **Remember:** SOLID principles help create maintainable, flexible, and testable code!

Single Responsibility Principle (SRP) - PYTHON

What is SRP?

"A class should have only one reason to change."

Each class should have a single responsibility and encapsulate only one part of the software's functionality.

Problems: This class has multiple responsibilities - account management, persistence, notifications, and reporting!

✗ Violation Example

```
class BankAccount:
    accountNumber: str
    customerName: str
    balance: float = 0.0
    _transactions: List[str] = field(default_factory=list)

    # --- Responsibility 1: Core Account Logic (Correct) ---
    def deposit(self, amount: float) -> None:
        self.balance += amount
        self._transactions.append(f"{datetime.now():%Y-%m-%d %H:%M} +{amount:.2f} -> {self.balance:.2f}")

    def withdraw(self, amount: float) -> None:
        self.balance -= amount
        self._transactions.append(f"{datetime.now():%Y-%m-%d %H:%M} -{amount:.2f} -> {self.balance:.2f}")

    # --- SRP VIOLATION 1: Persistence Logic ---
    def saveToDatabase(self) -> None: # stub for parity with Java
        pass

    def loadFromDatabase(self) -> None:
        pass

    # --- SRP VIOLATION 2: Notification Logic ---
    def sendEmailNotification(self) -> None:
        pass

    # --- SRP VIOLATION 3: Presentation/Reporting Logic ---
    def generateStatement(self) -> str:
        lines = [f"Statement for {self.customerName} ({self.accountNumber})",
                 "-" * 42]
        lines.extend(self._transactions)
        lines.append("-" * 42)
        lines.append(f"Current balance: {self.balance:.2f}")
        return "\n".join(lines)
```


SRP Violations Identified

● Violation 1: Persistence Logic

Methods: saveToDatabase(), loadFromDatabase()

Problem: What if the database changes from MySQL to MongoDB? The BankAccount class would need to change!

● Violation 2: Notification Logic

Methods: sendEmailNotification()

Problem: What if we want to send SMS instead of email? The BankAccount class would need to change!

● Violation 3: Presentation/Reporting Logic

Methods: generateStatement()

Problem: What if the statement format changes to PDF or we need different formats? The BankAccount class would need to change!

The Solution: Separate Responsibilities

✓ Refactored Design

- **BankAccount** - Only handles core account operations (deposit, withdraw, getBalance)
- **BankAccountRepository** - Handles database operations (save, load)
- **NotificationService** - Handles all notifications (email, SMS, push)
- **StatementGenerator** - Handles report generation (HTML, PDF, XML)

🎯 Benefits

- Each class has a single reason to change
- Easier to test individual components
- Better maintainability and extensibility
- Follows the Open-Closed Principle

Key Takeaways

✗ SRP Violations Lead To:

- Classes that are hard to maintain
- Changes ripple through unrelated functionality
- Difficult to test in isolation
- Tight coupling between different concerns

✓ Following SRP Results In:

- Highly cohesive classes
- Loosely coupled system
- Easy to extend and modify
- Better testability

"A class should have only one reason to change."



SRP_Violation.py




SRP_Complined.py

Open-Closed Principle

Strategy Pattern Solutions in Python

! Core Problem: The OCP Violation

The fundamental issue: The code must be **modified** every time a new requirement is added. This often happens with long `if/elif/else` or `switch` statements that check for a "type" and then execute logic.

 **Goal:** Add new functionality by writing **new code**, not by changing old, tested code.



Problematic Pattern

```
if (type.equals("SAVINGS")) {  
    // Savings logic  
    return balance * 0.02;  
} else if (type.equals("CHECKING")) {  
    // Checking logic  
    return balance * 0.001;  
} else if (type.equals("PREMIUM")) {  
    // Premium logic  
    return balance * 0.035;  
}  
// Every new type requires modifying this method!
```



The Solution: The Strategy Pattern

The most common solution is to use the **Strategy Pattern**. This involves three key steps:

- 1. Define a Contract:** Create a common interface or protocol that all variations of the logic will follow.
- 2. Implement Concrete Strategies:** Write separate classes for each variation, with each class implementing the contract.
- 3. Use the Contract:** Make the main "calculator" or "service" class work with the contract, not the concrete classes. This decouples the main logic from the specific implementations.

Approach in Python

Python is more flexible and often uses protocols and dictionaries for a dynamic approach.

Key Concept: Python uses "duck typing" and protocols for flexible, dynamic strategy implementation.



OCP_Violation.py



OCP_Complined.py

Step 1: Define Protocol Contract

This defines expected structure using "duck typing".

```
from typing import Protocol

class InterestStrategy(Protocol):
    def calculate(self, balance: float) -> float: ...
```

Step 2: Create Implementation Classes

No special 'implements' keyword needed with Protocol.

```
class SavingsStrategy:
    def calculate(self, balance: float) -> float:
        return balance * 0.02
```

Step 3: Use Dictionary Dispatch

A central service holds strategy mappings, avoiding if/elif chains.

```
# The service looks up the correct strategy and calls it.
rules = {"SAVINGS": SavingsStrategy()}
interest = rules["SAVINGS"].calculate(1000)
```



The Liskov Substitution Principle

Definition

The Liskov Substitution Principle is a core concept in object-oriented design that states if you have a class S that is a subtype of class T, you should be able to replace objects of type T with objects of type S without breaking the program.

In simple terms: A subclass should be a seamless replacement for its parent class.



Use Case: Personal Banking Management System

Let's model different types of bank accounts. We'll start with a base BankAccount class that has deposit and withdraw methods. We'll then create subtypes like SavingsAccount and a special FixedTermDepositAccount.



Key Concepts



What Causes Violations?

Violations often occur when a subclass overrides a parent method in a way that is unexpected, such as by throwing an exception it's not supposed to, doing nothing, or changing the fundamental behavior.



How to Spot Violations

Look for UnsupportedOperationException, empty implementations, or code that checks object types before calling methods.



The Core Problem

A FixedTermDepositAccount is a type of bank account, so inheriting from BankAccount seems logical. However, you cannot withdraw from a fixed-term account until it matures.



The Solution

Rethink the hierarchy! Create smaller, more specific interfaces. Use composition over inheritance when the "is-a" relationship doesn't truly fit.



LSP_Violation.py



LSP_Complied.py

LSP Violation Example - PYTHON



The Problem

```

class BankAccount:
    def __init__(self, balance: float = 0.0) -> None:
        self._balance = balance

    def deposit(self, amount: float) -> None:
        self._balance += amount

    def withdraw(self, amount: float) -> None:
        if self._balance < amount:
            raise ValueError("Insufficient funds")
        self._balance -= amount

    @property
    def balance(self) -> float:
        return self._balance

class SavingsAccount(BankAccount):
    # Inherits behavior as-is (OK)
    pass

class FixedTermDepositAccount(BankAccount):
    def withdraw(self, amount: float) -> None:
        # ✗ LSP violation: unexpected exception for a core behavior
        raise NotImplementedError("Cannot withdraw from a fixed-term deposit account.")

def process_withdrawal(acct: BankAccount, amount: float) -> None:
    # Expects any BankAccount to support withdraw()
    acct.withdraw(amount) # Will explode for FixedTermDepositAccount

if __name__ == "__main__":
    savings = SavingsAccount()
    savings.deposit(200)
    process_withdrawal(savings, 50) # OK

    fixed = FixedTermDepositAccount()
    fixed.deposit(500)
    try:
        process_withdrawal(fixed, 100) # ✗ NotImplementedError
    except Exception as e:
        print("Caught:", e)
    print("Balances -> savings:", savings.balance, "| fixed:", fixed.balance)

```



The Solution

```

class Account(ABC):
    def __init__(self, balance: float = 0.0) -> None:
        self._balance = balance
    def deposit(self, amount: float) -> None:
        self._balance += amount
    @property
    def balance(self) -> float:
        return self._balance

class Withdrawable(ABC):
    @abstractmethod
    def withdraw(self, amount: float) -> None:
        ...

class SavingsAccount(Account, Withdrawable):
    def withdraw(self, amount: float) -> None:
        if self._balance < amount:
            raise ValueError("Insufficient funds")
        self._balance -= amount

class FixedTermDepositAccount(Account):
    # No withdraw() - not withdrawable until maturity
    pass

def withdraw_if_possible(obj, amount: float) -> None:
    if isinstance(obj, Withdrawable):
        obj.withdraw(amount)
    else:
        print(f"Not withdrawable type: {obj.__class__.__name__}")

if __name__ == "__main__":
    savings = SavingsAccount()
    savings.deposit(300)
    withdraw_if_possible(savings, 120) # OK

    fixed = FixedTermDepositAccount()
    fixed.deposit(1000)
    withdraw_if_possible(fixed, 100) # Gracefully skipped

    print("Balances -> savings:", savings.balance, "| fixed:", fixed.balance)

```




Key Takeaways

🚩 How to Spot LSP Violations

- **UnsupportedOperationException:** A subclass method throws an exception the parent doesn't
- **Empty implementations:** A subclass method does nothing
- **Type checking:** Code checks object type before calling methods

✅ How to Fix Violations

- **"Tell, Don't Ask":** Don't ask for object type, tell it what to do
- **Rethink Hierarchy:** The "is-a" relationship might be wrong
- **Specific Interfaces:** Create role-based interfaces (Withdrawable, Depositable)

🏗️ Design Principles

- **Interface Segregation:** Break down large classes into smaller, focused interfaces
- **Composition over Inheritance:** Sometimes objects should contain behavior, not inherit it
- **Contract Compliance:** Subclasses must honor parent contracts

💎 Best Practices

- **Behavioral Compatibility:** Subclasses should strengthen postconditions, not weaken them
- **Precondition Rules:** Don't strengthen preconditions in subclasses
- **Semantic Consistency:** Maintain the meaning of operations across the hierarchy




Remember

"If it looks like a duck, quacks like a duck, but needs batteries — you probably have the wrong abstraction!"

Interface Segregation Principle

Building Clean, Focused Interfaces

"Clients should not be forced to depend on interfaces they do not use. It's better to have many small, specific interfaces than one large, general-purpose one."

 *It's like ordering a combo meal but being forced to take items you're allergic to. ISP lets you order just what you want.*



Core ISP Concepts



Core Idea

No client should be forced to implement interface methods it doesn't use

Avoid "fat interfaces" that bundle unrelated functionality

Keep interfaces focused and cohesive



Goal

Avoid forcing classes to implement irrelevant methods

Keep the system decoupled and clean

Improve maintainability and readability



Use Case: Banking System

Imagine a single, large `BankingOperations` interface with methods for every possible action: **`deposit()`**, **`withdraw()`**, **`openFixedDeposit()`**, **`applyForLoan()`**, **`requestCreditCardStatement()`**

❌ The Problem: ISP Violation - PYTHON

🚨 Fat Interface Violation

```
from typing import Protocol
```

```
class BankingOperations(Protocol):
    def check_balance(self, account_id: str) -> None: ...
    def make_deposit(self, account_id: str, amount: float) -> None: ...
    def make_withdrawal(self, account_id: str, amount: float) -> None: ...
    def open_new_account(self, customer_name: str) -> None: ...
    def close_account(self, account_id: str) -> None: ...

class AccountHolder(BankingOperations):
    def check_balance(self, account_id: str) -> None:
        print("Checking balance...")
    def make_deposit(self, account_id: str, amount: float) -> None:
        print("Depositing...")
    def make_withdrawal(self, account_id: str, amount: float) -> None:
        print("Withdrawing...")
    def open_new_account(self, customer_name: str) -> None:
        raise NotImplementedError("Account holders cannot open new accounts.")
    def close_account(self, account_id: str) -> None:
        raise NotImplementedError("Account holders cannot close accounts.")
```

✅ In summary:

The violation here is that `AccountHolder` is forced to implement operations (`openNewAccount` , `closeAccount`) that do not apply to its role, breaking the Interface Segregation Principle.

✖ Problem & Impact



Problems Created

Empty Methods: Forced to implement irrelevant functionality

Tight Coupling: Classes depend on methods they don't use

Poor Maintainability: Changes affect unrelated classes



Impact

Confusing Code: UnsupportedOperationException everywhere

Brittle Design: Adding features breaks existing code

Testing Complexity: Must test irrelevant methods



The Solution: Segregated Interfaces - PYTHON

Segregated Interfaces

CodeMate | Qodo Gen: Options | Test this class

```
class AccountHolderActions(Protocol):  
    def check_balance(self, account_id: str) -> None: ...  
    def make_deposit(self, account_id: str, amount: float) -> None: ...  
    def make_withdrawal(self, account_id: str, amount: float) -> None: ...
```

CodeMate | Qodo Gen: Options | Test this class

```
class BankTellerActions(Protocol):  
    def open_new_account(self, customer_name: str) -> None: ...  
    def close_account(self, account_id: str) -> None: ...  
    def make_deposit(self, account_id: str, amount: float) -> None: ...
```

Clean Implementation

CodeMate | Qodo Gen: Options | Test this class

```
class AccountHolderActions(Protocol):  
    def check_balance(self, account_id: str) -> None: ...  
    def make_deposit(self, account_id: str, amount: float) -> None: ...  
    def make_withdrawal(self, account_id: str, amount: float) -> None: ...
```

CodeMate | Qodo Gen: Options | Test this class

```
class AccountHolder:  
    Qodo Gen: Options | Test this method  
    def check_balance(self, account_id: str) -> None:  
        | print("Checking balance...")  
    Qodo Gen: Options | Test this method  
    def make_deposit(self, account_id: str, amount: float) -> None:  
        | print("Depositing...")  
    Qodo Gen: Options | Test this method  
    def make_withdrawal(self, account_id: str, amount: float) -> None:  
        | print("Withdrawing...")
```

CodeMate | Qodo Gen: Options | Test this function

```
def perform_customer_actions(client: AccountHolderActions, account_id: str):  
    | client.check_balance(account_id)
```



ISP_Violation.py



ISP_Complied.py

🌟 Benefits of ISP Compliance

🌟 High Cohesion

Classes only implement methods that are relevant to their purpose, making them more focused and easier to understand.

🌟 Low Coupling

Changes in one interface don't affect classes that don't use it. Loan changes won't require recompiling `SavingsAccount`.

🌟 Improved Readability

Clear separation of concerns makes the codebase easier to navigate and understand for new developers.

🌟 Enhanced Maintainability

The system becomes safer to change with reduced risk of breaking unrelated functionality.

🌟 Flexible Design

Classes can implement multiple small interfaces as needed, creating flexible and composable designs.

🌟 Better Testing

Focused interfaces lead to more targeted unit tests and easier mocking for test scenarios.



Dependency Inversion Principle

Creating Loosely Coupled & Resilient Software

Part 1

High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces).

Part 2

Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.



Simple Terms

Don't let important business logic (high-level) depend directly on specific, technical details (low-level). Instead, make both depend on a common contract or interface.



Python Solution: Using Protocols



DIP_Violation.py



DIP_Complined.py

✗ Why This Is Problematic

🔗 Tight Coupling

High-level policy (notifying about transactions) becomes tightly coupled to the low-level mechanism (sending an email).

♥ Rigidity

What if the bank decides to switch to SMS notifications or add push notifications? You would have to modify TransactionService every time.

🔧 Hard to Test

Cannot easily mock or substitute the notification mechanism for testing.

✗ Violating DIP

```
# Low-level module
class EmailNotifier:
    def send_email(self, message: str):
        print(f"Email: {message}")

# High-level module
class TransactionService:
    def __init__(self):
        # Direct dependency!
        self.notifier = EmailNotifier()

    def complete_transaction(self, amount: float):
        print(f"Transaction completed")
        self.notifier.send_email(f"Success!")
```

✓ Following DIP

```
from typing import Protocol

# Step 1: Create abstraction
class Notifier(Protocol):
    def send(self, message: str) -> None: ...

# Step 2: Concrete implementations
class EmailNotifier:
    def send(self, message: str) -> None:
        print(f"Email: {message}")

class SmsNotifier:
    def send(self, message: str) -> None:
        print(f"SMS: {message}")

# Step 3: Depend on abstraction
class TransactionService:
    def __init__(self, notifier: Notifier):
        self.notifier = notifier

    def complete_transaction(self, amount: float):
        print(f"Transaction completed")
        self.notifier.send(f"Success!")
```



Benefits of Following DIP



Why DIP Makes Your Code Better



Flexibility

Easily swap implementations (EmailNotifier → SmsNotifier) without changing business logic.



Maintainability

Changes are isolated and don't cascade through the system. Modify notification logic without touching transaction logic.



Testability

Easily "mock" dependencies (e.g., TestNotifier) to test TransactionService in isolation.



Loose Coupling

Modules are independent and can evolve separately. High-level policy is protected from low-level changes.



Extensibility

Add new notification types (PushNotifier, SlackNotifier) without modifying existing code.



Single Responsibility

Each class has a clear, focused purpose. Transaction logic and notification logic are separated.



Key Takeaway

"Depend upon abstractions, not concretions" - This principle inverts the typical dependency structure, making your architecture more flexible and maintainable.

DRY Principle

Don't Repeat Yourself

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

— The Pragmatic Programmer



Use Case: Personal Banking System

Customer Information Validation

Building a banking system where we need to validate customer information. A common requirement is ensuring email and phone number formats are correct before creating or updating customer profiles.



Single Source of Truth

One place to define validation logic



Easy Maintenance

Change once, apply everywhere



Fewer Bugs

No inconsistent implementations

❌ The Violation: Repeated Logic - PYTHON

🚩 Duplicated Validation Logic

```
class CustomerService:
    def createCustomer(self, name: str, email: str, phone: str) -> None:
        # Repeated validation block (❌)
        if not email or "@" not in email:
            raise ValueError("Invalid email format.")
        if not phone or len(phone) != 10:
            raise ValueError("Invalid phone number format.")
        print(f"Creating customer: {name}")
        # ... database logic to save new customer

    def updateCustomer(self, customerId: int, email: str, phone: str) -> None:
        # Repeated validation block (❌)
        if not email or "@" not in email:
            raise ValueError("Invalid email format.")
        if not phone or len(phone) != 10:
            raise ValueError("Invalid phone number format.")
        print(f"Updating customer: {customerId}")
        # ... database logic to update customer

if __name__ == "__main__":
    svc = CustomerService()
    svc.createCustomer("Alice", "alice@example.com", "1234567890")
    svc.updateCustomer(42, "bob@example.com", "0987654321")
```

Python



DRY_Violation.py



The Solution: Abstract the Logic - PYTHON

✦ Private Helper Method Approach

```
def validateContactInfo(email: str, phone: str) -> None:
    if not email or "@" not in email:
        raise ValueError("Invalid email format.")
    if not phone or len(phone) != 10:
        raise ValueError("Invalid phone number format.")

class CustomerService:
    def createCustomer(self, name: str, email: str, phone: str) -> None:
        validateContactInfo(email, phone) # ☒ single source of truth
        print(f"Creating customer: {name}")
        # ... database logic

    def updateCustomer(self, customerId: int, email: str, phone: str) -> None:
        validateContactInfo(email, phone) # ☒ single source of truth
        print(f"Updating customer: {customerId}")
        # ... database logic

# Optional: shared util module could hold validateContactInfo for use across services.

if __name__ == "__main__":
    svc = CustomerService()
    svc.createCustomer("Charlie", "charlie@example.com", "1112223333")
    svc.updateCustomer(7, "dana@example.com", "4445556666")
```

Python



DRY_Complied.py



Benefits of Following DRY



Maintainability

Change validation rules in one place, not scattered across multiple methods



Consistency

Identical logic everywhere prevents subtle bugs from variations



Testability

Test validation logic once, not in every method that uses it



Efficiency

Faster development and fewer lines of code to maintain

Key Takeaway

Identify → Abstract → Reuse

KISS Principle

Keep It Simple, Stupid

The KISS principle states that most systems work best if they are kept simple rather than made **complicated**. Simplicity should be a key goal in design, and unnecessary complexity should be avoided. In software, this means writing clear, straightforward, and easily understandable code.

Use Case: Personal Banking Management System

Requirement: Calculate a withdrawal fee. The rule is simple: a \$5 fee is applied if the account balance is less than \$500. Otherwise, the withdrawal is free.

The Violation (Over-engineered Code)

This code uses a dictionary to mimic a switch statement and includes redundant checks, which is not idiomatic or simple for this problem.

 VIOLATION: Overly complex for a simple binary choice 

```
# VIOLATION: Overly complex for a simple binary choice
def calculate_withdrawal_fee(balance: float, withdrawal_amount: float) -> float:
    fee_status = "NO_FEE"

    if withdrawal_amount > 0:
        if balance < 500:
            fee_status = "APPLY_FEE"

# Using a dictionary as a switch is overkill here
    fee_map = {
        "APPLY_FEE": 5.0,
        "NO_FEE": 0.0
    }

    return fee_map.get(fee_status, 0.0)
```



KISS_Violation.py

Python Solution Python

✓ The Solution (Simple & Pythonic)

This version is clean, readable, and uses a simple conditional expression, which is a common and clear pattern in Python.

✓ COMPLIANT: Simple, direct, and Pythonic

```
# COMPLIANT: Simple, direct, and Pythonic
def calculate_withdrawal_fee(balance: float) -> float:
    """Calculates a $5 withdrawal fee if the balance is below $500."""
    if balance < 500:
        return 5.0
    return 0.0

# An even more concise "one-liner" version
def calculate_withdrawal_fee_online(balance: float) -> float:
    """Calculates a $5 withdrawal fee if the balance is below $500."""
    return 5.0 if balance < 500 else 0.0
```



KISS_Complined.py

Python Key Takeaways Python



Key Improvements Made:

- **Use Direct Conditionals:** Replaced the complex dictionary lookup with a simple if/else.
- **Embrace Readability:** The simplified code reads almost like plain English.
- **Leverage Conditional Expressions:** Python's one-line if/else is perfect for simple binary choices.
- **Avoid Premature Generalization:** Don't build a complex structure (like the fee_map) for a simple problem. Solve the immediate problem in the simplest way possible.

Remember KISS

Simplicity is the Ultimate Sophistication

Simple code is:

- Easier to read and understand
- Faster to debug and maintain
- Less prone to bugs
- More testable
- Better for team collaboration



*"Perfection is achieved not when there is nothing more to add,
but when there is nothing left to take away."*

— Antoine de Saint-Exupéry



Refactoring a SalesReport Class

From Problematic Design to Professional Architecture

Learning Objectives

- ✦ Identify common design flaws
- 🔧 Apply SOLID principles
- 🏛️ Implement professional architecture patterns
- 🔑 Create testable, maintainable code

Let's inspect whether the class I formed is having issues or not. Go to file **SalesReport.py** . **Solution, We will do an exercise after the Design principle lectures**

! The Problematic Class

A SalesReport class that *works* but has serious architectural flaws:

```
class SalesReport:
    Qodo Gen: Options | Test this method
    def __init__(self, user): ...

    Qodo Gen: Options | Test this method
    def generate(self): ...

    Qodo Gen: Options | Test this method
    def export_to_json(self): ...

    Qodo Gen: Options | Test this method
    def get_raw_data(self): ...

# --- Example Usage ---
CodeMate
report = SalesReport("admin_user")
print(report.generate())

CodeMate
raw_data_ref = report.get_raw_data()
raw_data_ref.clear()
```

 **Key Point:** This code works, but it's rigid, hard to test, and difficult to maintain in real-world applications.



Five Critical Design Flaws

1. Hardcoded Dependencies

Database connection string is hardcoded. Impossible to test without a running MySQL database.

2. Single Responsibility Violation

One method handles data fetching, processing, AND formatting. Multiple reasons to change.

3. Logic Duplication

JSON export duplicates calculation logic. Changes must be made in multiple places.

4. Exposed Internal State

External code can modify internal data list, leading to unpredictable behavior.

5. Implicit Dependencies

Data structure format is not formally defined. Changes break multiple places.



The Solution Architecture

Separation of Concerns

Data Sources

MySQLDataSource
FileDataSource
APIDataSource

Report Model

SalesReport
Data Processing
Business Logic

Formatters

PlainTextFormatter
JsonFormatter
CsvFormatter

Key Benefits

- ✓ Each class has a single, clear responsibility
- ✓ Easy to test individual components
- ✓ Can swap implementations without changing other parts
- ✓ Adding new formats or data sources is simple



Step 1: Define Clear Data Structures

```
from dataclasses import dataclass

@dataclass
class SaleItem:
    """A simple, immutable data container for a sale record."""
    item: str
    quantity: int
    price: float

    @property
    def revenue(self) -> float:
        return self.quantity * self.price
```

✨ Benefits of @dataclass

- **Self-documenting:** Clear structure definition
- **Immutable:** Data integrity protection
- **Type hints:** Better IDE support and error catching
- **Built-in methods:** `__init__`, `__repr__`, `__eq__` automatically generated




Step 2: Dependency Injection

```
from abc import ABC, abstractmethod

class IDataSource(ABC):
    """Interface for any data source that can provide sales data."""
    @abstractmethod
    def get_sales_data(self) -> List[SaleItem]:
        pass

class MySqlDataSource(IDataSource):
    def __init__(self, connection_string: str):
        self._connection_string = connection_string

    def get_sales_data(self) -> List[SaleItem]:
        print(f"Connecting to {self._connection_string}...")
        return [
            SaleItem(item="Laptop", quantity=2, price=1200),
            SaleItem(item="Mouse", quantity=5, price=25)
        ]
```

 **Key Benefit:** The report no longer creates its own database connection. We can easily inject different data sources for testing or different environments.



Step 3: Single Responsibility

```
class SalesReport:
    """
    Processes sales data but does NOT handle fetching or formatting.
    Its single responsibility is holding and calculating report data.
    """

    def __init__(self, user: str, source: IDataSource):
        self.user = user
        self._source = source
        self._items: List[SaleItem] = self._source.get_sales_data()

    @property
    def total_revenue(self) -> float:
        """Calculates total revenue from the items."""
        return sum(item.revenue for item in self._items)

    def get_items(self) -> List[SaleItem]:
        """Return a copy to prevent external modification."""
        return self._items.copy()
```

✗ Before

One method handled: data fetching, processing, AND formatting

✓ After

Class only manages report data and calculations



Step 4: Flexible Formatting System

```
import json
from abc import ABC, abstractmethod

# Assuming SalesReport and SaleItem classes are defined elsewhere
# from your_module import SalesReport, SaleItem

class IReportFormatter(ABC):
    """Interface for any class that can format a SalesReport."""
    @abstractmethod
    def format(self, report: 'SalesReport') -> str:
        pass

class PlainTextFormatter(IReportFormatter):
    """Formats the report into a human-readable plain text string."""
    def format(self, report: 'SalesReport') -> str:
        content = f"Sales Report for {report.user}\n"
        content += "-----\n"
        for item in report.get_items():
            content += f"- Item: {item.item}, Revenue: ${item.revenue:,.2f}\n"
        content += f"Total Revenue: ${report.total_revenue:,.2f}"
        return content

class JsonFormatter(IReportFormatter):
    """Formats the report into a JSON string."""
    def format(self, report: 'SalesReport') -> str:
        output = {
            "report_user": report.user,
            "total_revenue": report.total_revenue,
            "items": [item.__dict__ for item in report.get_items()]
        }
        return json.dumps(output, indent=2)
```



Putting It All Together

```
# 1. Choose your data source
mysql_source = MySQLDataSource("mysql://user:pass@localhost/sales")

# 2. Create the report object, injecting the dependency
report = SalesReport("admin_user", source=mysql_source)

# 3. Choose your desired output format and format the report
text_formatter = PlainTextFormatter()
json_formatter = JsonFormatter()

print("--- Plain Text Report ---")
print(text_formatter.format(report))

print("\n--- JSON Report ---")
print(json_formatter.format(report))
```

Notice the Flexibility

- Want to test? Inject a MockDataSource
- Need CSV export? Create a CsvFormatter
- Switch to PostgreSQL? Create a PostgreSQLDataSource
- All without modifying existing classes!



Summary of Improvements



Technical Benefits

- **Testable:** Easy to mock dependencies
- **Maintainable:** Changes isolated to specific classes
- **Extensible:** Add new features without modification
- **Reusable:** Components can be used independently



Business Benefits

- **Faster development:** Clear responsibilities
- **Lower risk:** Changes don't break other features
- **Team collaboration:** Multiple developers can work simultaneously
- **Future-proof:** Easy to adapt to new requirements



Key Takeaway

"Good software design isn't about writing code that works—it's about writing code that works and can adapt to change gracefully."

? Questions & Discussion

Think About

- How would you add email reporting?
- What if we needed database transactions?
- How would you handle errors?

Next Steps

- Practice with your own classes
- Look for similar patterns in your code
- Start small, refactor incrementally

Thank you!



Pydantic vs. ORM Models

A Decoupled Approach for Robust APIs

Use Case: Personal Banking Management System

Key Takeaway: Separate your concerns for robust and maintainable APIs

Core Concepts - What Are They?

Python



Pydantic Models

The "API Layer"

Purpose: Define the shape of external data. They act as the public-facing contract for your API.

Key Function: Data validation, serialization (Python to JSON), and deserialization (JSON to Python).

Analogy: Think of them as a customs declaration form ensuring everything entering or leaving your application is well-formed, typed, and valid.



ORM Models

The "Database Layer"

Purpose: Define the structure of your database tables. They map Python objects directly to database rows.

Key Function: Database interaction (Create, Read, Update, Delete - CRUD operations).

Analogy: Think of them as the blueprint for a bank vault defining exactly how data is structured and stored securely.

Why Use Both? The Power of Decoupling

Python

Keeping Pydantic and ORM models separate is a best practice for several key reasons:

Security

You never expose your internal database structure directly to the outside world. Your API can have a different shape than your database tables, preventing accidental data leaks.

Flexibility

Your API can evolve independently of your database schema. You can add a field to your API response without needing to add a new column to your database, and vice-versa.

Clarity

Each model has a single, clear responsibility. Pydantic models handle API validation. ORM models handle database state.

Validation Control

You can have different validation rules for incoming data (e.g., password must be 8 characters) versus what's stored in the database (e.g., a hashed password).

The Pattern - How They Interact

Python

The typical workflow follows a clear, one-way data flow, managed by a "Service" or "Business Logic" layer.

1 API Endpoint (/create_customer)

Receives raw JSON data from an HTTP request. Uses a Pydantic model (CustomerCreate) to parse, validate, and type-cast this incoming data.

2 Service Layer (customer_service.py)

Receives the validated Pydantic model object. Contains the business logic (e.g., check if email exists, hash the password). Maps the data from the Pydantic model to an ORM model (CustomerDB).

3 Database Layer (crud.py)

Receives the populated ORM model object. Commits the ORM model to the database, creating a new row.

4 Response

The newly created data from the database (in the ORM model) is converted back into a Pydantic model (CustomerPublic) to be sent as a JSON response, ensuring sensitive data (like password hashes) is excluded.

Use Case - Creating a New Bank Customer

Python

Let's apply this pattern to adding a new customer to our Personal Banking Management System.

Requirement:

An API endpoint to create a new customer with a name, email, and password. The password must be stored securely, and the API response should not include the password.

Pydantic Models

The API Contract

CustomerCreate: For incoming data. Requires name, email, and a password.

CustomerPublic: For outgoing data. Includes id, name, and email, but omits the password.

ORM Model

The Database Blueprint

CustomerDB: Represents the customers table. Has columns for id, name, email, and hashed_password.

Code Example - Putting It All Together

Python

1. Pydantic Models (schemas.py)

```
from pydantic import BaseModel, EmailStr

# For incoming data -> The "Create" schema
class CustomerCreate(BaseModel):
    name: str
    email: EmailStr
    password: str

# For outgoing data -> The "Public" schema
class CustomerPublic(BaseModel):
    id: int
    name: str
    email: EmailStr

class Config:
    from_attributes = True # Allows creating from ORM model
```



schemas.py

2. ORM Model (models.py)

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

# Represents the database table
class CustomerDB(Base):
    __tablename__ = "customers"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
```



models.py

Service Layer Implementation

Python

3. Service Layer Logic (service.py)

```
# Fictional hashing function
def get_password_hash(password: str) -> str:
    return f"hashed_{password}"

# Business logic that connects the two models
def create_customer(customer_data: CustomerCreate) -> CustomerDB:
    # 1. Hash the password from the Pydantic model
    hashed_password = get_password_hash(customer_data.password)

    # 2. Create an ORM model instance
    db_customer = CustomerDB(
        name=customer_data.name,
        email=customer_data.email,
        hashed_password=hashed_password
    )

    # 3. Save db_customer to the database (not shown)
    # db.add(db_customer)
    # db.commit()
    # db.refresh(db_customer)

    return db_customer
```



service.py

Key Point: The service layer acts as the bridge between your API and database, handling business logic and data transformation.



Dependency Injection (DI)

Building Flexible & Testable Code

The "Hollywood Principle":
"Don't call us, we'll call you" - Inversion of Control (IoC)

Core Idea: A design pattern where a class receives its dependencies from an external source rather than creating them itself.

Analogy:

Think of a car. A car doesn't build its own engine. The engine (a dependency) is built separately and is injected into the car during assembly.

Main Goal:

To achieve **Loose Coupling** between classes, making your code more modular, flexible, and easier to test.

It's a form of a broader principle called Inversion of Control (IoC).

Understanding DI Through Analogy

Think of a Chef in a Kitchen

✗ Without DI

The chef has to:

- 🌱 Grow their own vegetables
- 🐄 Raise their own cattle
- ⌚ Do everything from scratch every time
- 🔧 Manage the entire supply chain

Result: Chef is overwhelmed and can't focus on cooking!

✓ With DI

A supplier delivers:

- 🌱 Fresh ingredients (dependencies)
- 🍖 Quality meat products
- 📦 Everything ready to use
- 🔄 Easy to swap suppliers if needed

Result: Chef focuses solely on cooking excellence!

Why Use Dependency Injection?



Decoupling

Classes are not tightly bound to their dependencies. A service that sends notifications doesn't need to know *how* the notification is sent (Email, SMS, etc.), only that it has a "notifier" that can send a message.



Enhanced Testability

You can easily "mock" or fake dependencies during testing. To test a transaction service, you can inject a `MockNotificationService` that prints to console instead of sending real emails or SMS.



Flexibility & Maintainability

Easy to swap out components. If your bank decides to switch from emails to SMS messages, you just inject a different notification service without changing core transaction logic.

The Problem: Banking System Without DI 👉

Python

✗ Tightly Coupled Dependencies

The TransactionService creates its own dependencies directly, leading to rigid and fragile code.

✗ Bad Practice - Tightly Coupled Code

```
# Tightly coupled dependencies
class EmailNotifier:
    def send(self, message: str):
        print(f"EMAIL Sent: {message}")

class TransactionService:
    def __init__(self):
        # The service CREATES its own dependency. This is bad!
        self._notifier = EmailNotifier()

    def process_transaction(self, amount: float):
        # ... logic to process the transaction ...
        self._notifier.send(f"Processed transaction of {amount}")

# Usage
service = TransactionService()
service.process_transaction(100.0)
```

Problems with This Approach:

- 🔒 **Testing Issues:** You can't test TransactionService without also testing the real EmailNotifier. No mock objects possible!
- 🚫 **Inflexibility:** Want to switch to SMS? You must change the TransactionService code itself.

The Solution: Banking System With DI 👍

Python

✓ Loosely Coupled with Constructor Injection

Dependencies are injected through the constructor, making the code flexible and testable.

✓ Good Practice - Dependency Injection

```
# Define the dependencies
class EmailNotifier:
    def send(self, message: str):
        print(f"EMAIL Sent: {message}")

class SmsNotifier:
    def send(self, message: str):
        print(f"SMS Sent: {message}")

# --- The Refactored Service ---
class TransactionService:
    def __init__(self, notifier):
        # The dependency is INJECTED, not created.
        self._notifier = notifier

    def process_transaction(self, amount: float):
        # ... logic to process the transaction ...
        self._notifier.send(f"Processed transaction of {amount}")
```

Another Method: Setter Injection

Dependencies are provided through public setter methods after the object has been created.

Loosely Coupled with Setter Injection

```
public class AccountService {  
  
    private AccountRepository repository;  
    private NotificationService notificationService;  
  
    // Default constructor is empty.  
    public AccountService() {}  
  
    // Dependencies are "injected" via setter methods.  
    public void setRepository(AccountRepository repository) {  
        this.repository = repository;  
    }  
  
    public void setNotificationService(NotificationService notificationService) {  
        this.notificationService = notificationService;  
    }  
  
    // ... createAccount method ...  
}
```

✓ Use Case:

Best for optional dependencies that are not critical for the object's initial state.

⚠ Drawback:

The object can exist in an incomplete state before its dependencies are set.

Methods of Dependency Injection

Python

Constructor Injection (Preferred)

When: Dependencies are passed through the class constructor

Best for: Required dependencies

Benefit: Ensures object is created in valid state

```
class AccountService:
    def __init__(self, repo, notifier):
        self.repo = repo
        self.notifier = notifier
```

Setter Injection

When: Dependencies are set via setter methods after creation

Best for: Optional dependencies

Drawback: Object can exist in incomplete state

```
class AccountService:
    def set_repository(self, repo):
        self.repo = repo

    def set_notifier(self, notifier):
        self.notifier = notifier
```

Putting It All Together (The "Injector")

An external part of your application is responsible for creating dependencies and injecting them.

Putting it all together — The "Injector"

```
public class DI_PersonalBankingDemo {  
    public static void main(String[] args) {  
        System.out.println("=== Tight Coupling (avoid) ===");  
        TightlyCoupledAccountService bad = new TightlyCoupledAccountService();  
        bad.createAccount("Ria Sharma");  
  
        System.out.println("\n=== Constructor Injection (preferred) ===");  
        // 1) Create dependencies (can be switched without changing AccountService)  
        AccountRepository repo = new DatabaseAccountRepository();  
        NotificationService email = new EmailNotificationService();  
        // NotificationService sms = new SmsNotificationService(); // easy swap  
        // 2) Inject into the service  
        AccountService good = new AccountService(repo, email);  
        // 3) Use the service  
        good.createAccount("Ria Sharma");  
  
        System.out.println("\n=== Setter Injection ===");  
        AccountServiceWithSetters withSetters = new AccountServiceWithSetters();  
        withSetters.setRepository(new DatabaseAccountRepository());  
        withSetters.setNotificationService(new SmsNotificationService()); // show easy swap  
        withSetters.createAccount("Ria Sharma");  
  
        System.out.println("\n=== Testing with a Mock (benefit of DI) ===");  
        MockNotificationService mock = new MockNotificationService();  
        AccountService testable = new AccountService(new DatabaseAccountRepository(), mock);  
        testable.createAccount("Test User");  
        System.out.println("Mock captured messages: " + mock.getSentMessages());  
    }  
}
```



DI_PersonalBankingDemo.java

1

Create all required dependencies first

2

Inject dependencies into the main service

3

Use the fully configured service



💡 Flexible Usage - Same Service, Different Dependencies

```
# --- Flexible Usage ---  
# Injecting an EmailNotifier  
email_notifier = EmailNotifier()  
email_service = TransactionService(notifier=email_notifier)  
email_service.process_transaction(250.0)  
# Output: EMAIL Sent: Processed transaction of 250.0  
  
# Injecting an SmsNotifier without changing the service!  
sms_notifier = SmsNotifier()  
sms_service = TransactionService(notifier=sms_notifier)  
sms_service.process_transaction(500.0)  
# Output: SMS Sent: Processed transaction of 500.0  
  
# For testing - Mock notifier  
class MockNotifier:  
    def send(self, message: str):  
        print(f"MOCK: {message}")  
  
mock_notifier = MockNotifier()  
test_service = TransactionService(notifier=mock_notifier)  
test_service.process_transaction(1000.0)  
# Output: MOCK: Processed transaction of 1000.0
```



dependency_injection_banking.py

🔑 **Key Insight:** The same TransactionService works with different notifiers without any code changes to the service itself!

Key Takeaways

Decoupling

Classes are independent. AccountService doesn't know or care which NotificationService it's using, as long as it fulfills the contract.

Easier Testing

You can easily "inject" mock or fake objects during tests. For example, provide a MockNotificationService to verify method calls without actually sending emails.

Improved Reusability

Components can be easily swapped and reused in different contexts. Same service, different implementations.

Better Maintainability

Code is cleaner, more organized, and easier to understand and modify. You change configurations, not code.

Remember: Dependency Injection promotes the principle of "Don't call us, we'll call you" - let the framework or container manage your dependencies!



Building APIs in Python

The Python Developer's Guide



light_banking_api.zip

Python

Implementing Spring Patterns with FastAPI, Pydantic & SQLAlchemy

Goal: Show that while syntax and libraries change, the principles of building clean, layered APIs are universal

FastAPI

Spring Boot equivalent

Pydantic

Bean Validation + DTOs

SQLAlchemy

JPA/Hibernate equivalent



Today's Challenge (in 30 Minutes!)

Your Mission: Implement the "Create Bank Account" feature

Build a complete feature from API endpoint down to the database, exactly as you would in a layered Java application



Controller/API Layer

Receives HTTP requests
Handles validation & routing



Service Layer

Executes business logic
Orchestrates operations



Persistence Layer

Interacts with database
Manages data operations

Feature Specification:

- **API Endpoint:** POST /accounts/
- **Request Data:** account_type, initial balance, customer_id
- **Business Logic:** Generate unique account number
- **Database:** Save to bank_accounts table
- **Response:** Return complete account details with new ID



The Core Pattern: Python vs. Java

The pattern is identical. Only the names of the tools change.

| Concept | Java / Spring Boot | Python / FastAPI |
|----------------------|-----------------------------|-----------------------------|
| API Data Carrier | DTO (Plain Old Java Object) | Schema (Pydantic BaseModel) |
| Database Model | @Entity (JPA) | ORM Model (SQLAlchemy) |
| Business Logic | @Service Class | Plain Python Class |
| API Endpoint | @RestController | Path Operation Function |
| Dependency Injection | @Autowired | Depends() |

Key Takeaway: If you understand layered architecture in Java, you already understand it in Python. The concepts are universal!



Step 1: The Data Contract (Schemas / DTOs)

A Pydantic Schema defines the "shape" of your API's JSON data. It's the direct equivalent of a Java DTO class used with `@RequestBody` and for responses.

Input Schema (AccountCreate)

For the request body

Output Schema (Account)

For the response body

```
# In schemas.py (like a dto package)
```

```
from pydantic import BaseModel
```

```
# Equivalent to an AccountCreateDTO.java
```

```
class AccountCreate(BaseModel):
```

```
    account_type: str
```

```
    balance: float
```

```
    customer_id: int
```

```
# Equivalent to an AccountResponseDTO.java
```

```
class Account(BaseModel):
```

```
    id: int
```

```
    account_number: str
```

```
    account_type: str
```

```
    balance: float
```

```
    customer_id: int
```

```
class Config:
```

```
    # This tells Pydantic to be compatible with ORM models
```

```
    # It's like using a mapping library (e.g., MapStruct)
```

```
    orm_mode = True
```

Step 2: The Database Model (ORM Models / Entities)

An SQLAlchemy ORM Model represents a row in a database table. It's the direct equivalent of a JPA @Entity class.

```
# In models.py (like a domain or entity package)
```

```
from sqlalchemy import Column, Integer, String, Float, ForeignKey
from .database import Base

# Equivalent to a @Entity class
class BankAccount(Base):
    __tablename__ = "bank_accounts" # like @Table(name="...")

    id = Column(Integer, primary_key=True) # like @Id
    account_number = Column(String, unique=True, index=True, nullable=False)
    account_type = Column(String, nullable=False)
    balance = Column(Float, nullable=False)
    customer_id = Column(Integer, ForeignKey("customers.id")) # like @ManyToOne

# Dummy Customer model for the foreign key relationship
class Customer(Base):
    __tablename__ = "customers"
    id = Column(Integer, primary_key=True)
    name = Column(String)
```

Java Equivalent: This is exactly like defining a JPA Entity with @Entity, @Table, @Id, @Column, and @ManyToOne annotations.



Step 3: The "Brain" (Service Layer)

The Service Class contains the core business logic. It is decoupled from the web and database specifics. It's the direct equivalent of a Spring @Service class.

```
# In services.py (like a service package)
```

```
from sqlalchemy.orm import Session
from . import models, schemas
import random

class AccountService:
    # The DB session is injected, like injecting a Repository in Spring
    def __init__(self, db: Session):
        self.db = db

    def create_account(self, account_schema: schemas.AccountCreate) -> models.BankAccount:
        # 1. Business Logic: Generate a unique account number
        account_number = f"ACC{random.randint(1000000, 9999999)}"

        # 2. Map from Schema (DTO) to ORM Model (Entity)
        db_account = models.BankAccount(
            account_type=account_schema.account_type,
            balance=account_schema.balance,
            customer_id=account_schema.customer_id,
            account_number=account_number
        )

        # 3. Save to DB (this would be repository.save() in Spring)
        self.db.add(db_account)
        self.db.commit()
        self.db.refresh(db_account)
        return db_account
```

Java Equivalent: This is like a @Service class with @Autowired Repository dependency and business logic methods.



Step 4: Tying It All Together (Controller & DI)

The Path Operation Function is FastAPI's equivalent of a Spring @RestController method. Its job is to handle HTTP interaction and delegate to the service.

In main.py (like a controller package)

```
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from . import models, schemas, services
from .database import engine, get_db

# Create database tables
models.Base.metadata.create_all(bind=engine)

app = FastAPI()

@app.post("/accounts/", response_model=schemas.Account, status_code=201)
def create_new_account(
    account: schemas.AccountCreate,      # 1. Equivalent to @RequestBody
    db: Session = Depends(get_db)       # 2. DI, equivalent to @Autowired
):
    # 3. Basic validation (would be in service layer in larger apps)
    customer = db.query(models.Customer).filter(
        models.Customer.id == account.customer_id
    ).first()
    if not customer:
        raise HTTPException(
            status_code=404,
            detail=f"Customer with id {account.customer_id} not found."
        )

    # 4. Create service and delegate
    service = services.AccountService(db=db)
    return service.create_account(account_schema=account)
```

Java Equivalent: This is like a @RestController with @PostMapping, @RequestBody, and @Autowired dependencies.

Database Configuration & Dependency Injection

Setting up the database connection and DI mechanism (equivalent to Spring Boot's auto-configuration)

```
# In database.py
```

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# Use SQLite database for this example
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL,
    connect_args={"check_same_thread": False}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()

# This function is the dependency provider.
# FastAPI's Depends() will call this for each request.
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Java Equivalent: This is like Spring Boot's DataSource configuration with @Configuration and @Bean annotations for database connectivity.



Let's Get Building!

Your 30-Minute Challenge Starts Now!

1

Define your Pydantic Schemas

Create `schemas.py` with `AccountCreate` and `Account` models (equivalent to DTOs)



light_banking_api.zip

2

Create your SQLAlchemy ORM Model

Create `models.py` with `BankAccount` and `Customer` entities

3

Implement the AccountService

Create `services.py` with business logic (equivalent to `@Service`)

4

Create the FastAPI endpoint

Update `main.py` with the POST endpoint (equivalent to `@RestController`)

Pro Tip: Focus on the patterns, not the syntax. The architecture is the same as Java Spring Boot!



Conclusion & Key Takeaways

The Pattern is Universal

Controller → Service → Persistence is a robust, standard architecture in any modern backend framework. The concepts of DTOs/Schemas (for API data) and Entities/Models (for DB tables) are fundamental to clean, decoupled design.



Dependency Injection is Your Best Friend

DI (FastAPI's **Depends** or Spring's **@Autowired**) is the mechanism that enables this clean, layered architecture. It makes our code testable and maintainable.



You've Learned a New Stack, Not a New Paradigm

If you understand how to build a layered application in Java, you now understand how to do it in Python. **The principles are the same.**

What You Built Today

- Clean API endpoints with FastAPI
- Data validation with Pydantic
- Business logic in service layers
- Database operations with SQLAlchemy
- Dependency injection patterns

Skills Transferred from Java

- Layered architecture design
- Separation of concerns
- DTO/Entity patterns
- Dependency injection concepts
- RESTful API principles

Thanks