1. **Single Responsibility**:

A class should have one and only one reason to change, meaning that a class should have only one job.

Ex: Imagine a **banking system** where a BankAccount class handles **account balance, transactions, and notifications**.

**Bad Example (Without SRP):**

|  |
| --- |
| class BankAccount {  private double balance;    public void deposit(double amount) {  balance += amount;  System.out.println("Deposited: " + amount);  }  public void withdraw(double amount) {  balance -= amount;  System.out.println("Withdrawn: " + amount);  }  public void generateStatement() {  System.out.println("Generating statement for account");  }  public void sendSMSNotification(String message) {  System.out.println("Sending SMS: " + message);  }  } |

**Why is this bad?**

🚨 BankAccount is doing too much!

* **Handles balance and transactions**
* **Generates bank statements**
* **Sends SMS notifications**

**Good Example (With SRP Applied):**

|  |
| --- |
| // Separate class for balance and transactions  class BankAccount {  private double balance;    public void deposit(double amount) {  balance += amount;  System.out.println("Deposited: " + amount);  }  public void withdraw(double amount) {  balance -= amount;  System.out.println("Withdrawn: " + amount);  }  } |

|  |
| --- |
| // Separate class for statements  class StatementGenerator {  public void generateStatement() {  System.out.println("Generating statement for account");  }  } |

|  |
| --- |
| // Separate class for SMS notifications  class SMSNotifier {  public void sendSMS(String message) {  System.out.println("Sending SMS: " + message);  }  } |

**Why is this better?**

✅ BankAccount only **manages account balance**.

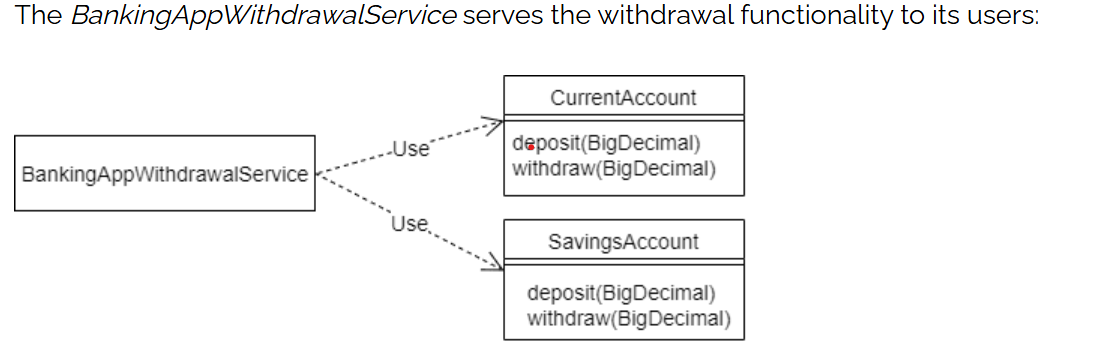
✅ StatementGenerator **generates statements** separately.

✅ SMSNotifier **handles notifications independently**.

1. **Open/Closed Principle**

It states that our Class/Module should be open for extension and close for Modification.

Ex: Banking application supports two types of account Savings and Current.



Public class WithdrawalService {

@Autowired

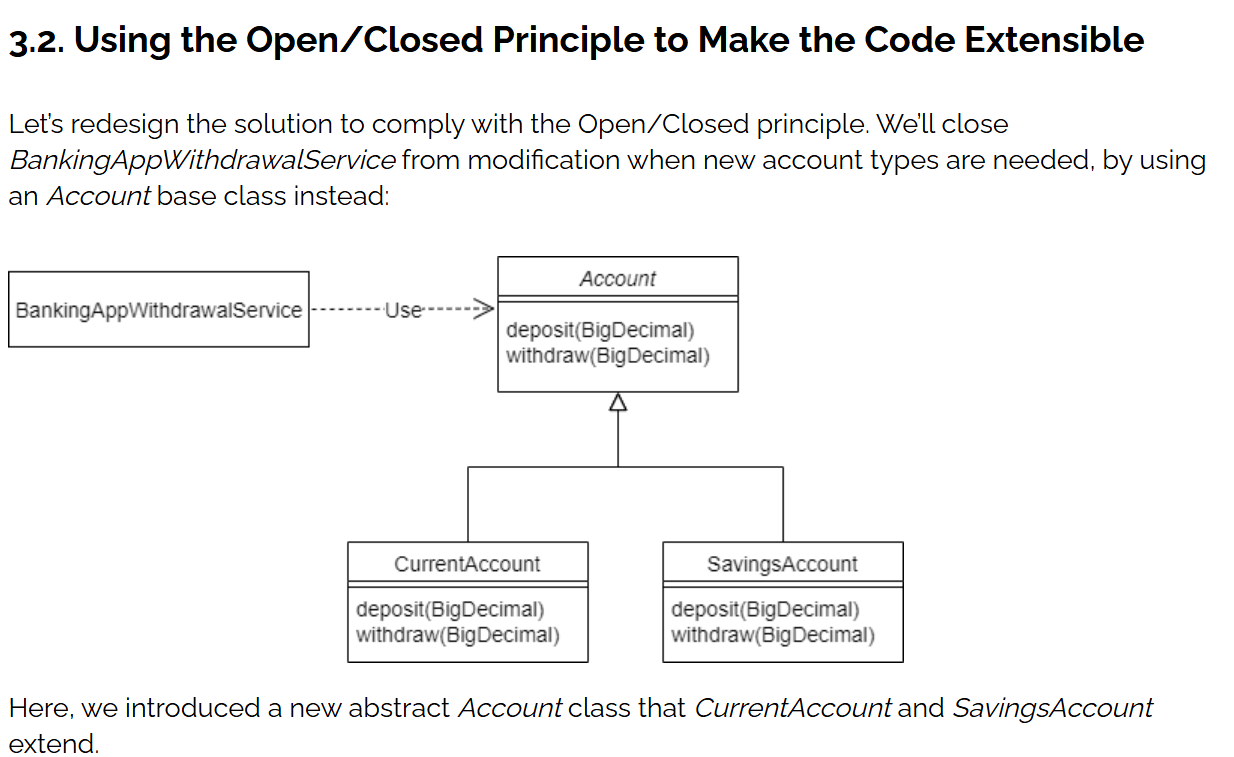
Private CurrentAccount currentAccount;

Private SavingsAccount savingsAccount;

// currentAccount.withdraw();

// savingsAccount.withdraw();

Unfortunately, there is a problem with extending this design. The WithdrawalService is aware of the two concrete implementations of account*.* Therefore, the WithdrawalService would need to be changed every time a new account type is introduced.



The *WithdrawalService* no longer depends on concrete account classes. Because it now depends only on the abstract class, it need not be changed when a new account type is introduced.

Consequently, the *WithdrawalService* is **open for extension** with new account types, but **closed for modification**, in that the new types don’t require it to change in order to integrate.

Let’s look at this example in Java. To begin with, let’s define the *Account* class:

**public** **abstract** **class** **Account** {

**protected** **abstract** **void** **deposit** (BigDecimal amount);

/\*\*

\* Reduces the balance of the account by the specified amount

\* provided given amount > 0 and account meets minimum available

\* balance criteria.

\*

\* **@param** amount

\*/

**protected** **abstract** **void** **withdraw** (BigDecimal amount);

}

And, let’s define the *WithdrawalService*:

**public** **class** **WithdrawalService** {

**private** Account account;

**public** **WithdrawalService**(Account account) {

this.account = account;

}

**public** **void** **withdraw**(BigDecimal amount) {

account.withdraw(amount);

}

}

}

Now, let’s look at how, in this design, a new account type might violate the Liskov Substitution Principle.

**3.4. A New Account Type**

The bank now wants to offer a high interest-earning fixed-term deposit account to its customers.

To support this, let’s introduce a new *FixedTermDepositAccount* class. A fixed-term deposit account in the real world “is a” type of account. This implies inheritance in our object-oriented design.

So, let’s make *FixedTermDepositAccount* a subclass of *Account*:

**public** **class** **FixedTermDepositAccount** **extends** **Account** {

// Overridden methods...

}

So far, so good. However, the bank doesn’t want to allow withdrawals for the fixed-term deposit accounts.

This means that the new *FixedTermDepositAccount* class can’t meaningfully provide the *withdraw* method that *Account* defines. One common workaround for this is to make *FixedTermDepositAccount* throw an *UnsupportedOperationException* in the method it cannot fulfill:

**public** **class** **FixedTermDepositAccount** **extends** **Account** {

@Override

**protected** **void** **deposit**(BigDecimal amount) {

// Deposit into this account

}

@Override

**protected** **void** **withdraw**(BigDecimal amount) {

**throw** **new** **UnsupportedOperationException**("Withdrawals are not supported by FixedTermDepositAccount!!");

}

}

**3.5. Testing Using the New Account Type**

While the new class works fine, let’s try to use it with the *BankingAppWithdrawalService*:

**Account** myFixedTermDepositAccount = **new** **FixedTermDepositAccount**();

myFixedTermDepositAccount.deposit(**new** **BigDecimal**(1000.00));

**BankingAppWithdrawalService** withdrawalService = **new** **BankingAppWithdrawalService**(myFixedTermDepositAccount);

withdrawalService.withdraw(**new** **BigDecimal**(100.00));Copy

Unsurprisingly, the banking application crashes with the error:

Withdrawals are not supported by FixedTermDepositAccount!!

There’s clearly something wrong with this design if a valid combination of objects results in an error.

**3.6. What Wrong?**

The *BankingAppWithdrawalService* is a client of the *Account* class. It expects that both *Account* and its subtypes guarantee the behavior that the *Account* class has specified for its *withdraw* method:

/\*\*

\* Reduces the account balance by the specified amount

\* provided given amount > 0 and account meets minimum available

\* balance criteria.

\*

\* **@param** amount

\*/

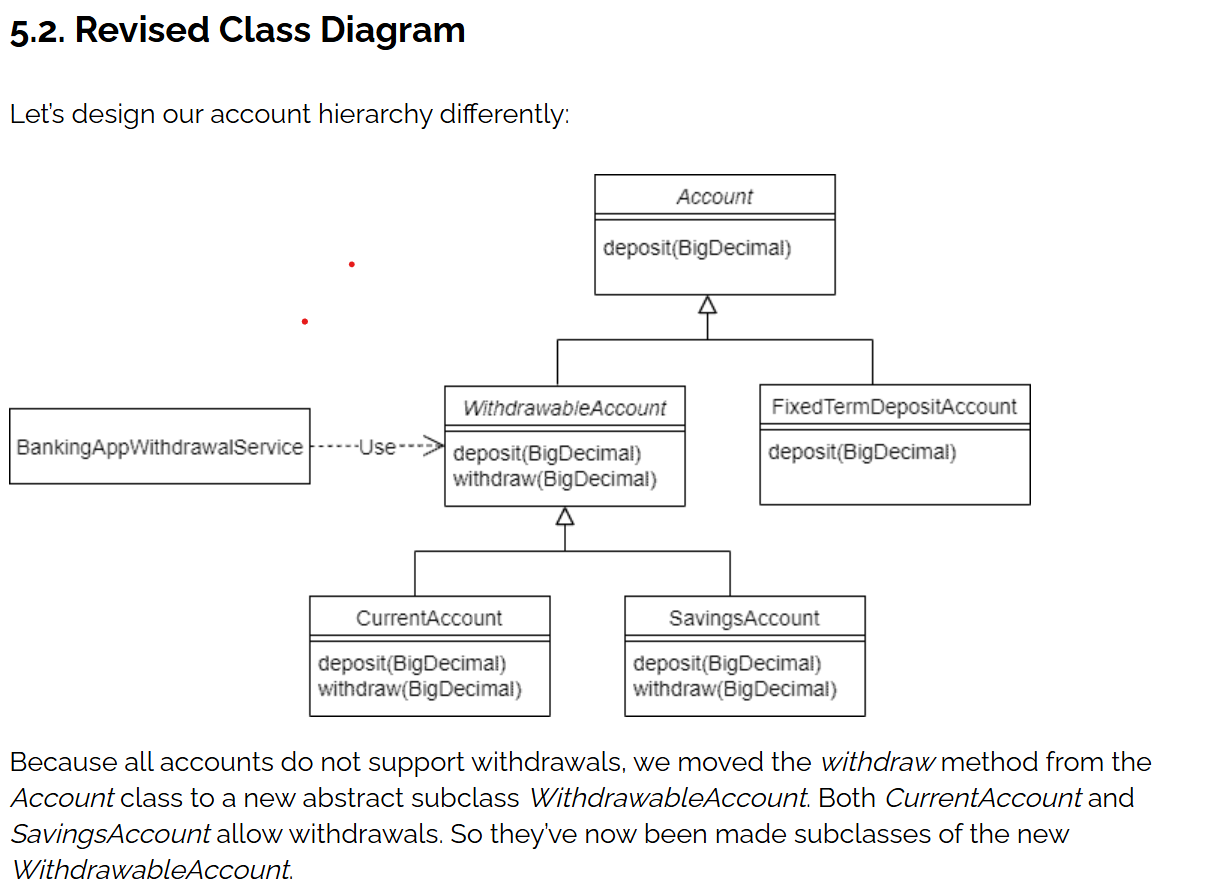
**protected** **abstract** **void** **withdraw**(BigDecimal amount);

However, by not supporting the *withdraw* method, the *FixedTermDepositAccount* violates this method specification*.* Therefore, we cannot substitute *FixedTermDepositAccount* for *Account*.

In other words, the *FixedTermDepositAccount* has violated the Liskov Substitution Principle.

1. **The Liskov Substitution Principle**

*Subtypes must be substitutable for their base types.*

**

***5.3. Refactored BankingAppWithdrawalService***

***BankingAppWithdrawalService* now needs to use the *WithdrawableAccount:***

|  |
| --- |
| **public class BankingAppWithdrawalService {**  **private WithdrawableAccount withdrawableAccount;**  **public BankingAppWithdrawalService(WithdrawableAccount withdrawableAccount) {**  **this.withdrawableAccount = withdrawableAccount;**  **}**  **public void withdraw(BigDecimal amount) {**  **withdrawableAccount.withdraw(amount);**  **}**  **}** |

As for *FixedTermDepositAccount*, we retain *Account* as its parent class. Consequently, it inherits only the *deposit* behavior that it can reliably fulfill and no longer inherits the *withdraw* method that it doesn’t want. This new design avoids the issues we saw earlier.

1. **Interface Segregation Principle**

**❌ Bad Design (Violates ISP)**

It states that no client should be forced to depend on methods or interfaces it does not use.

interface MediaPlayer {

void playAudio();

void playVideo();

}

class AudioPlayer implements MediaPlayer {

public void playAudio() {

System.out.println("Playing audio...");

}

public void playVideo() {

// This player doesn't support video

throw new UnsupportedOperationException("Video not supported");

}

}

class VideoPlayer implements MediaPlayer {

public void playAudio() {

System.out.println("Playing audio...");

}

public void playVideo() {

System.out.println("Playing video...");

}

}

✅ **Good Design (Follows ISP)**

interface AudioMediaPlayer {

void playAudio();

}

interface VideoMediaPlayer {

void playVideo();

}

class AudioPlayer implements AudioMediaPlayer {

public void playAudio() {

System.out.println("Playing audio...");

}

}

class VideoPlayer implements AudioMediaPlayer, VideoMediaPlayer {

public void playAudio() {

System.out.println("Playing audio...");

}

public void playVideo() {

System.out.println("Playing video...");

}

}

1. **Dependency Inversion Principle**

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

Let’s start by **defining the high-level component**:

|  |
| --- |
| **public** **class** **CustomerService**  {  **private** **final** CustomerDao customerDao; // standard constructor / getter  **public** Optional<Customer> **findById**(**int** id)  {  **return** customerDao.findById(id);  }  **public** List<Customer> **findAll**()  {  **return** customerDao.findAll();  }  } |

As we can see, the *CustomerService* class implements the *findById()* and *findAll()* methods, which fetch customers from the persistence layer using a simple [DAO](https://www.baeldung.com/java-dao-pattern) implementation.

Of course, we could’ve encapsulated more functionality in the class, but let’s keep it like this for simplicity’s sake.

In this case, **the *CustomerDao* type is the abstraction**that *CustomerService* uses for consuming the low-level component.

Since this a direct DIP implementation, let’s define the abstraction as an interface in the same package of *CustomerService*:

|  |
| --- |
| **public** **interface** **CustomerDao**  {  Optional<Customer> **findById**(**int** id);  List<Customer> **findAll**();  } |

By placing the abstraction in the same package of the high-level component, we’re making the component responsible for owning the abstraction. This implementation detail is **what really inverts the dependency between the high-level component and the low-level one**.

In addition, **the level of abstraction of *CustomerDao* is close to the one of *CustomerService****,* which is also required for a good DIP implementation.

Now, let’s create the low-level component in a different package. In this case, it’s just a basic *CustomerDao* implementation:

|  |
| --- |
| **public** **class** **SimpleCustomerDao** **implements** **CustomerDao** { // standard constructor / getter  @Override **public** Optional<Customer> **findById**(**int** id)  { **return** Optional.ofNullable(customers.get(id));  }  @Override **public** List<Customer> **findAll**() {  **return** **new** **ArrayList**<>(customers.values());  }  } |