

SC/CZ/CE2002 Object-Oriented Design & Programming

Chapter 9: Design Principles

Dr. Li Fang
Senior Lecturer, CCDS

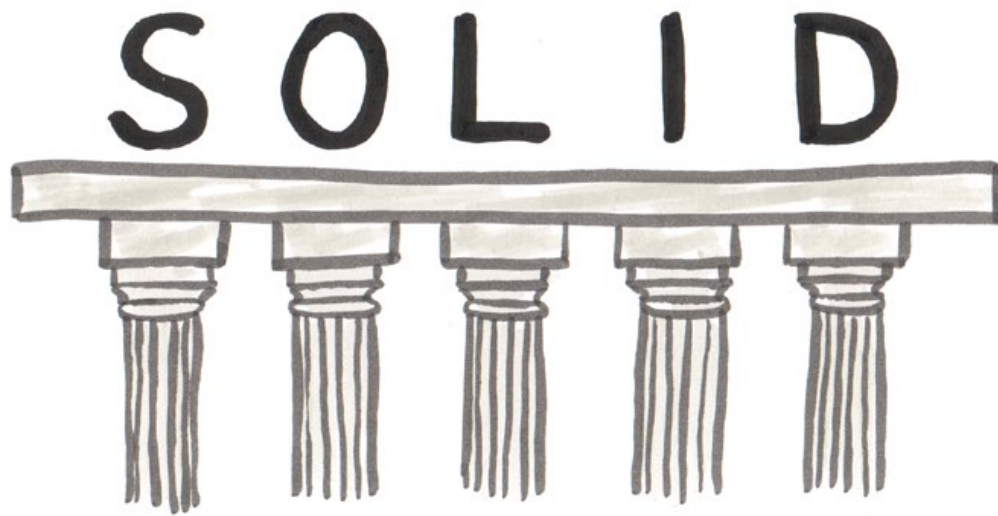


**NANYANG
TECHNOLOGICAL
UNIVERSITY**
SINGAPORE

Objectives

By the end of this chapter, you should be able to:

- Explain the symptoms of rotting design
- Describe the various OO design goals
- Explain SOLID design principles





TOPIC

Topic 1: OO Design Goals

OO Design Goals

- Make software **easier to change** when we want to.
 - We might want to change **a class or package** to add new functionality, change business rules or improve the design.
 - We might have to change a class or package because of a change to **another class or package it depends on** (e.g., a change to a method signature).
 - **Manage dependencies** between classes and packages of classes to **minimise impact** of change on other parts of the software.
 - Minimise reasons that modules or packages might be forced to change because of a change in a module or package it **depends** upon.

OO Design Goals

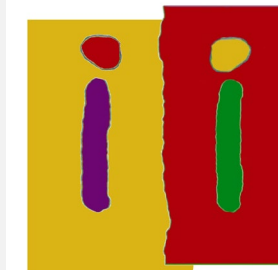
- Much of OO design is about [managing dependencies](#).
- It is very difficult to write OO code without creating a dependency on something.



Class Design Guidelines

Manage

- Coupling and Cohesion
 - Design with **Reuse (Reusability)** in mind
 - Design with **Extensibility** in mind
 - Design with **Maintainability** in mind
- To achieve Loose (Low) Coupling and High Cohesion



International
Independent



TOPIC

Topic 2: SOLID Design Principles

SRP

Single Responsibility Principle

"A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE."

Gather together the things that change for the same reasons. Separate those things that change for different reasons.

S

OCP

Open/Closed Principle

"SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS etc) SHOULD BE OPEN FOR EXTENSION BUT CLOSED FOR MODIFICATION"

O

LSP

Liskov Substitution Principle

"SUBCLASSES SHOULD BEHAVE NICELY WHEN USED IN PLACE OF THEIR BASE CLASS"

The sub-types must be replaceable for super-types without breaking the program execution.

L

ISP

Interface Segregation Principle

"A CLIENT SHOULD NEVER BE FORCED TO IMPLEMENT AN INTERFACE THAT IT DOESN'T USE OR CLIENTS SHOULDN'T BE FORCED TO DEPEND ON METHODS THEY DON'T USE"

Keep protocols small, don't force classes to implement methods they can't.

I

DIP

Dependency Inversion Principle

"HIGH-LEVEL MODULES SHOULD NOT DEPEND ON LOW-LEVEL MODULES. BOTH SHOULD DEPEND ON ABSTRACTIONS. ABSTRACTIONS SHOULD NOT DEPEND ON DETAILS. DETAILS SHOULD DEPEND ON ABSTRACTIONS"

D

Single Responsibility Principle (SRP)

“There should never be more than ONE reason for a class to change”.

- = cohesion
- class should be having one and only one responsibility



- You can avoid these problems by asking a simple question before you make any changes: What is the responsibility of your class/component/microservice?
- If your answer includes the word “and”, you’re most likely breaking the single responsibility principle. Then it’s better to take a step back and rethink your current approach. There is most likely a better way to implement it.

```
class Book {  
    String name;  
    String authorName;  
    int year;  
    int price;  
    String isbn;  
  
    public Book(String name, String authorName, int year, int price, String isbn) {  
        this.name = name;  
        this.authorName = authorName;  
        this.year = year;  
        this.price = price;  
        this.isbn = isbn;  
    }  
}
```

```

public class Invoice {

    private Book book;
    private int quantity;
    private double discountRate;
    private double taxRate;
    private double total;

    public Invoice(Book book, int quantity, double discountRate, double taxRate) {
        this.book = book;
        this.quantity = quantity;
        this.discountRate = discountRate;
        this.taxRate = taxRate;
        this.total = this.calculateTotal();
    }

    public double calculateTotal() {
        double price = ((book.price - book.price * discountRate) * this.quantity);

        double priceWithTaxes = price * (1 + taxRate);

        return priceWithTaxes;
    }

    public void printInvoice() {
        System.out.println(quantity + "x " + book.name + " " + book.price + "$");
        System.out.println("Discount Rate: " + discountRate);
        System.out.println("Tax Rate: " + taxRate);
        System.out.println("Total: " + total);
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }

}

```

slido






Which of the following are the possible reasons to change the code of Invoice class (multiple correct answers)?

① Start presenting to display the poll results on this slide.

Which of the following is the possible reason to change Invoice class?

Violates Single Responsibility Principle (SRP)

- A. The printing format changes 
- B. The invoice will be saved into database instead of a file 
- C. The formula to calculate discount changes 
- D. A new book is published.



According to SRP there should be three classes each having the single responsibility.

```
public class InvoicePrinter {
    private Invoice invoice;

    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }

    public void print() {
        System.out.println(invoice.quantity + "x " +
invoice.book.name + " " + invoice.book.price + " $");
        System.out.println("Discount Rate: " +
invoice.discountRate);
        System.out.println("Tax Rate: " + invoice.taxRate);
        System.out.println("Total: " + invoice.total + " $");
    }
}
```

Open-Closed Principle (OCP)

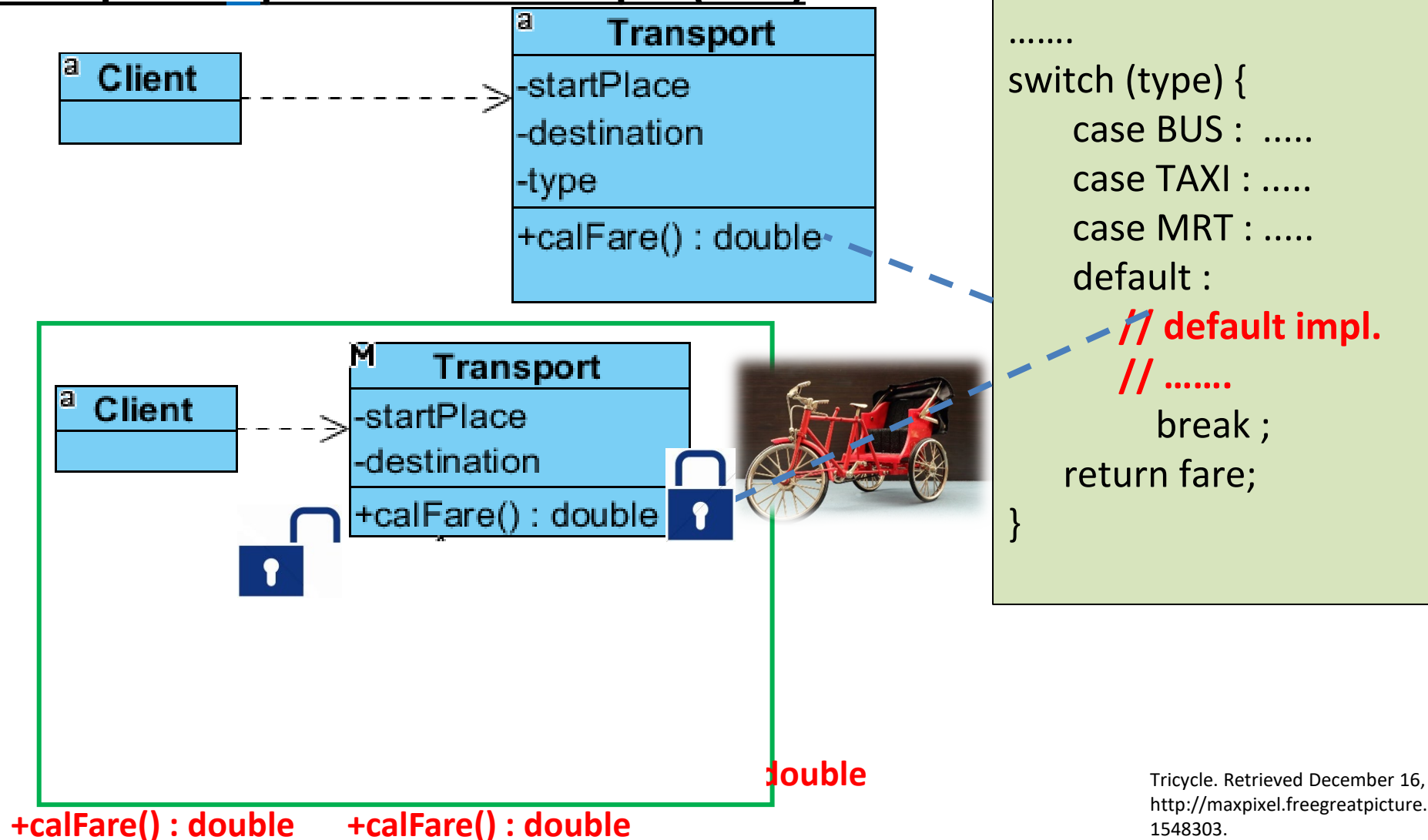
“A module should be open for extension but closed for modification”.

- if the class A is written by the developer AA, and if the developer BB wants some modification on that then developer BB should be easily do that by extending class A, but not by modifying class A.
- ***Abstraction is the key to the OCP.***

Extend but do not Change

SOLID Design Principles

Example of Open-Closed Principle (OCP)



```
public interface CalculatorOperation {}
```

```
public class Addition implements  
CalculatorOperation {  
    private double left;  
    private double right;  
    private double result = 0.0;  
  
    public Addition(double left, double right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    // getters and setters  
}
```

```
public class Subtraction implements  
CalculatorOperation {  
    private double left;  
    private double right;  
    private double result = 0.0;  
  
    public Subtraction(double left, double right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    // getters and setters  
}
```

```
public class Calculator {  
  
    public void calculate(CalculatorOperation operation) {  
        if (operation == null) {  
            throw new InvalidParameterException("Can not perform operation");  
        }  
  
        if (operation instanceof Addition) {  
            Addition addition = (Addition) operation;  
            addition.setResult(addition.getLeft() + addition.getRight());  
        } else if (operation instanceof Subtraction) {  
            Subtraction subtraction = (Subtraction) operation;  
            subtraction.setResult(subtraction.getLeft() - subtraction.getRight());  
        }  
    }  
}
```


slido



The above classes

① Start presenting to display the poll results on this slide.

When a new requirement of adding multiplication or division functionality comes in, any solution?

We've no way but to change the calculate method of the Calculator class.

It violates OCP

```
public interface CalculatorOperation {  
    void perform();  
}
```

```
public class Addition implements  
    CalculatorOperation {  
    private double left;  
    private double right;  
    private double result;  
  
    // constructor, getters and setters  
  
    @Override  
    public void perform() {  
        result = left + right;  
    }  
}
```

```
public class Division implements CalculatorOperation {  
    private double left;  
    private double right;  
    private double result;  
  
    // constructor, getters and setters  
    @Override  
    public void perform() {  
        if (right != 0) {  
            result = left / right;  
        }  
    }  
}
```

```
public class Calculator {  
    public void calculate(CalculatorOperation operation) {  
        if (operation == null) {  
            throw new InvalidParameterException("Cannot perform  
operation");  
        }  
        operation.perform();  
    }  
}
```

Our Calculator class doesn't need to implement new logic as we introduce new operators. That way the class is closed for modification but open for an extension.

WHAT IS THE OUTPUT OF THE FOLLOWING PROGRAM?

```
public class Animal {  
    public void makeNoise(){  
        System.out.print("Some sound");  
    }  
}  
  
class Dog extends Animal{  
  
    public void makeNoise(){  
        System.out.print("Bark");  
    }  
}  
  
class Cat extends Animal{  
  
    public void makeNoise(){  
        System.out.print("Meawoo");  
    }  
}
```


```
public class Test {  
    public static void main(String[] args) {  
        Animal a1 = new Cat();  
        a1.makeNoise();  
        Animal a2 = new Dog();  
        a2.makeNoise();  
    }  
}
```

- A. Some soundSome sound
- B. Some soundBark
- C. MeawooBark
- D. Some soundMeawooSome soundBark

WHAT IS THE OUTPUT OF THE FOLLOWING PROGRAM?

```
public class Animal {  
    public void makeNoise(){  
        System.out.print("Some sound");  
    }  
}  
  
class Dog extends Animal{  
  
    public void makeNoise(){  
        System.out.print("Bark");  
    }  
}  
  
class Cat extends Animal{  
  
    public void makeNoise(){  
        System.out.print("Meawoo");  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Animal a1 = new Cat();  
        a1.makeNoise();  
        Animal a2 = new Dog();  
        a2.makeNoise();  
    }  
}
```

- A. Some soundSome sound
- B. Some soundBark
-  C. MeawooBark
- D. Some soundMeawooSome soundBark

SOLID Design Principles

Liskov Substitution Principle (LSP)

“if for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T .”

Class	Class Type	Object	substitution
T	superclass	o_2	o_1
S	subclass	o_1	

- If class B is a subtype of class A, we should be able to replace A with B without disrupting the behavior of our program.
- The child class should not implement code such that if it is replaced by the parent class then the application will stop running.

Liskov Substitution Principle (LSP)

Subclass must do all the things super class do

Subclass must not bring any trouble that super class don't

WHAT IS THE OUTPUT OF THE FOLLOWING PROGRAM?

```
public class Animal {  
    public void makeNoise(){  
        System.out.print("Some sound");  
    }  
}  
  
class Dog extends Animal{  
  
    public void makeNoise(){  
        System.out.print("Bark");  
    }  
}  
  
class Cat extends Animal{  
  
    public void makeNoise(){  
        System.out.print("Meawoo");  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Animal a1 = new Cat();  
        a1.makeNoise();  
        Animal a2 = new DumbDog();  
        a2.makeNoise();  
    }  
}
```

```
class DumbDog extends Animal {  
  
    public void makeNoise() {  
        throw new RuntimeException("I can't  
make noise");  
    }  
}
```

**violates Liskov Substitution
Principle (LSP)**

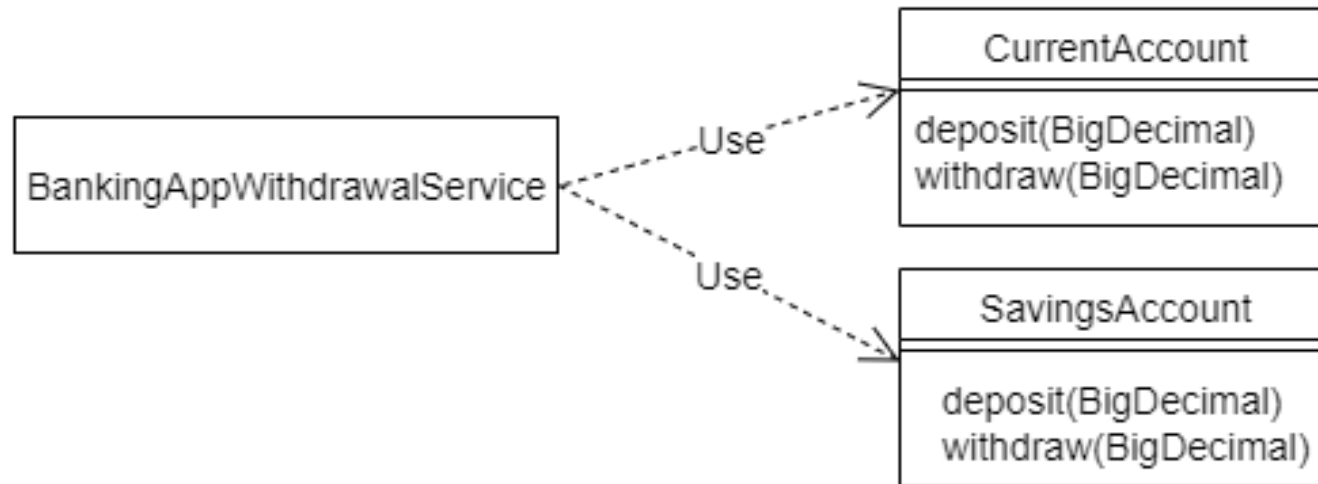
slido



Please analyse the following design and give your comments.

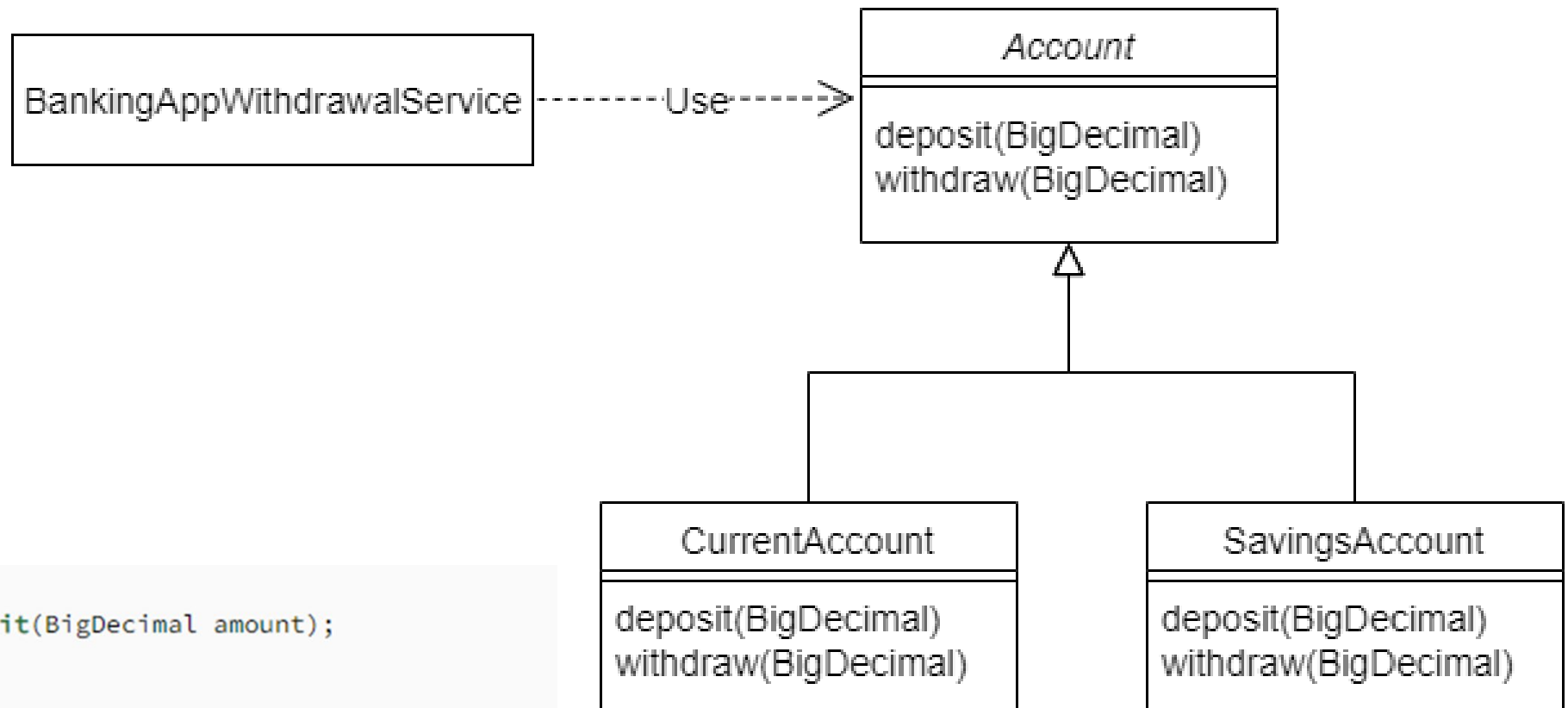
① Start presenting to display the poll results on this slide.

The following design



the `BankingAppWithdrawalService` would need to be changed every time a new account type is introduced

- A. looks fine.
- B. violates Single Responsibility Principle (SRP)
- ✓ C. violates Open-Closed Principle (OCP)
- D. violates Liskov Substitution Principle (LSP)



```
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);
}
```

The bank now wants to offer a high interest-earning fixed-term deposit account to its customers. However, **the bank doesn't want to allow withdrawals for the fixed-term deposit accounts.**

```
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!!");  
    }  
}
```

slido



Please analyse the design of the fixed-term deposit account and give your comments.

① Start presenting to display the poll results on this slide.

The above design

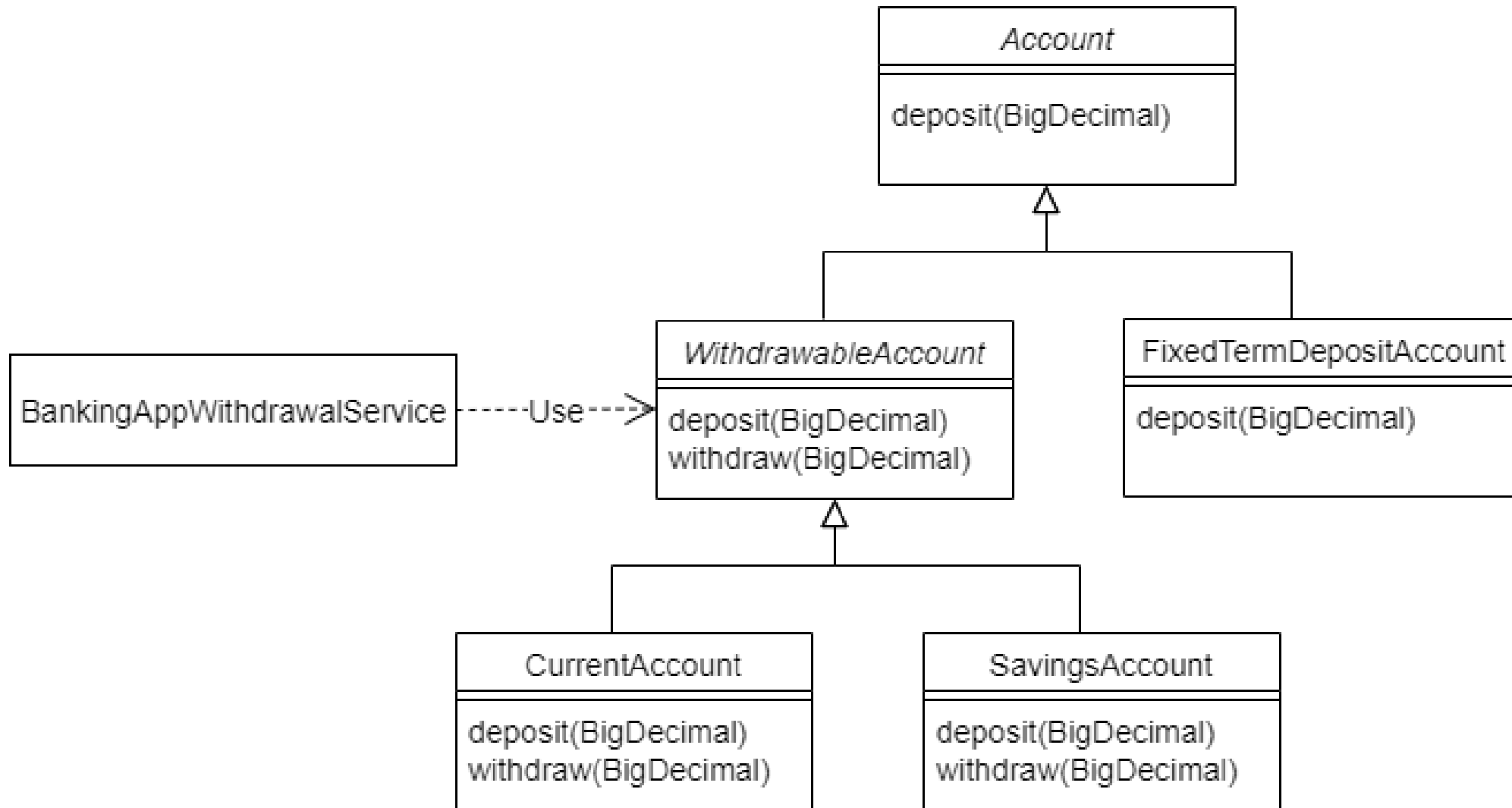
- A. look fine.
- B. violates Single Responsibility Principle (SRP)
- C. violates Open-Closed Principle (OCP)
- ✓ D. violates Liskov Substitution Principle (LSP)

```
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!!");  
    }  
}
```

```
Account myFixedTermDepositAccount = new  
FixedTermDepositAccount();  
myFixedTermDepositAccount.deposit(new BigDecimal(1000.00));  
BankingAppWithdrawalService withdrawalService = new  
BankingAppWithdrawalService(myFixedTermDepositAccount);  
withdrawalService.withdraw(new BigDecimal(100.00));
```

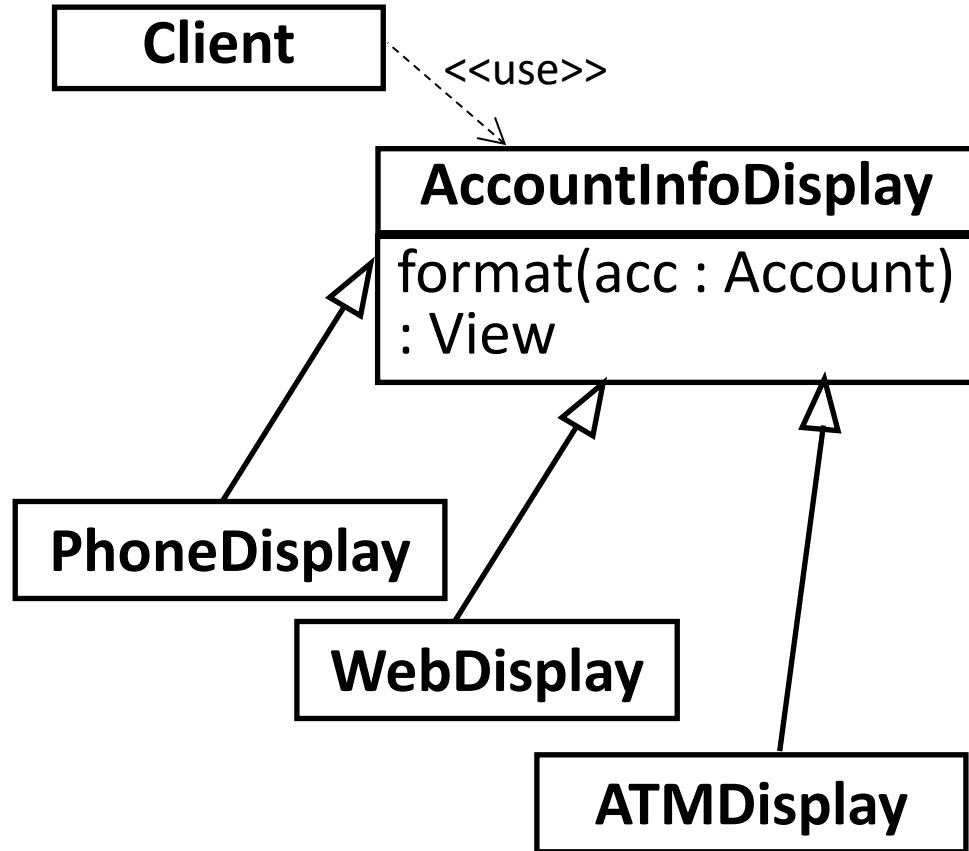


the banking application crashes with the error: **Withdrawals are not supported by FixedTermDepositAccount!!**



SOLID Design Principles

Example of **Liskov** Substitution Principle



```
public class AccountInfoDisplay {
    .....
    public View format(Account acc) {
        if (acc == null) return ;
        // codes to format and return full
        // details of acc View
    }
}
```

```
public class ATMDisplay extends
AccountInfoDisplay {
    .....
    public View format(Account acc) {
        // assume client check for acc not
        null
        // codes to format and return
        //partial details of acc View
    }
}
```

Example of Liskov Substitution Principle

```
public class AccountInfoDisplay {  
    .....  
    public View format(Account acc) {  
        if (acc == null) return ;  
        // codes to format and return full  
        // details of acc View  
    }  
}
```

```
public class ATMDisplay extends  
AccountInfoDisplay {  
    .....  
    public View format(Account acc) {  
        // assume client check for acc not  
        null  
        // codes to format and return  
        //partial details of acc View  
    }  
}
```

	superClass	subClass
Error checking		

Example of Liskov Substitution Principle

```
public class AccountInfoDisplay {  
    .....  
    public View format(Account acc) {  
        if (acc == null) return ;  
        // codes to format and return full  
        // details of acc View  
    }  
}
```

```
public class ATMDisplay extends  
AccountInfoDisplay {  
    .....  
    public View format(Account acc) {  
        // assume client check for acc not  
        null  
        // codes to format and return  
        //partial details of acc View  
    }  
}
```

	superClass	subClass
Error checking	if (acc == null) return ;	N/A

Example of Liskov Substitution Principle

```
public class AccountInfoDisplay {  
    .....  
    public View format(Account acc) {  
        if (acc == null) return ;  
        // codes to format and return full  
        // details of acc View  
    }  
}
```

```
public class ATMDisplay extends  
AccountInfoDisplay {  
    .....  
    public View format(Account acc) {  
        // assume client check for acc not  
        null  
        // codes to format and return  
        //partial details of acc View  
    }  
}
```

	superClass	subClass
Error checking	if (acc == null) return ;	N/A
Return information		

Example of Liskov Substitution Principle

```
public class AccountInfoDisplay {  
    .....  
    public View format(Account acc) {  
        if (acc == null) return ;  
        // codes to format and return full  
        // details of acc View  
    }  
}
```

```
public class ATMDisplay extends  
AccountInfoDisplay {  
    .....  
    public View format(Account acc) {  
        // assume client check for acc not  
        null  
        // codes to format and return  
        //partial details of acc View  
    }  
}
```

	superClass	subClass
Error checking	if (acc == null) return ;	N/A
Return information	full details of acc View	partial details of acc View

Liskov Substitution Principle (LSP)

Subclass must do all the things super class do

Subclass must not bring any trouble that super class don't

Interface Segregation Principle (ISP)

“Many client specific interfaces are better than one general purpose interface”.

- ***larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.***
- Avoid FAT Interfaces.
- This is the first principle which is applied on interface, all the above three principles applies on classes.



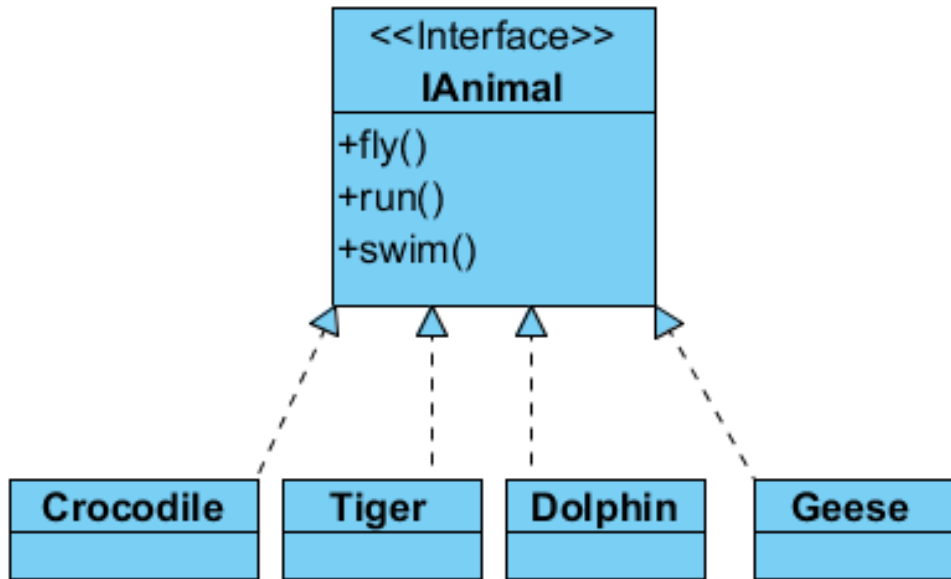
slido



According to the following class diagram, Tiger class needs implement which method to be a concrete class?

① Start presenting to display the poll results on this slide.

According to the following class diagram, Tiger class needs implement which method to be a concrete class?



- A. fly()
- B. run()
- C. swim()
- D. None of above
- ✓ E. All of above

Clients should not be forced to depend on methods in interfaces that they don't use.

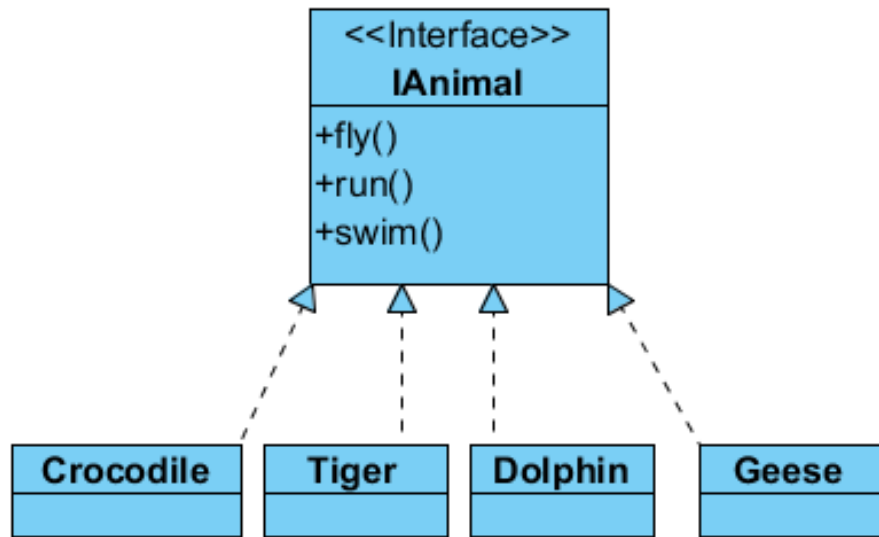
The goal of this principle is to reduce the side effects of using larger interfaces by breaking application interfaces into smaller ones. It's similar to the Single Responsibility Principle, where each class or interface serves a single purpose.

Precise application design and correct abstraction is the key behind the Interface Segregation Principle. **Though it'll take more time and effort in the design phase of an application and might increase the code complexity, in the end, we get a flexible code.**

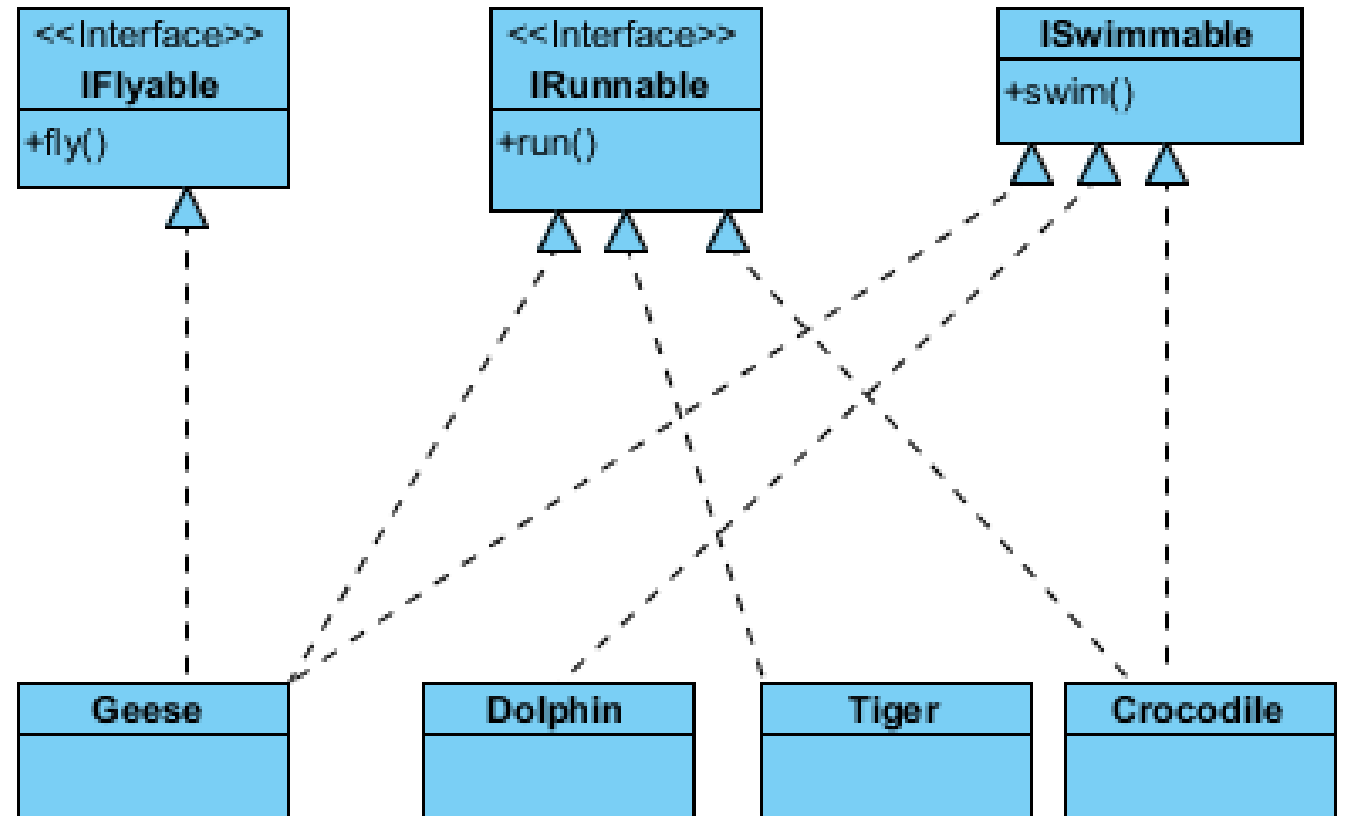
SOLID Design Principles

Interface Segregation Principle (ISP)

Separate Interface
when necessary



```
public class Tiger implements IAnimal{
    public void fly { }
    public void run { // code added }
    public void swim { }
}
```

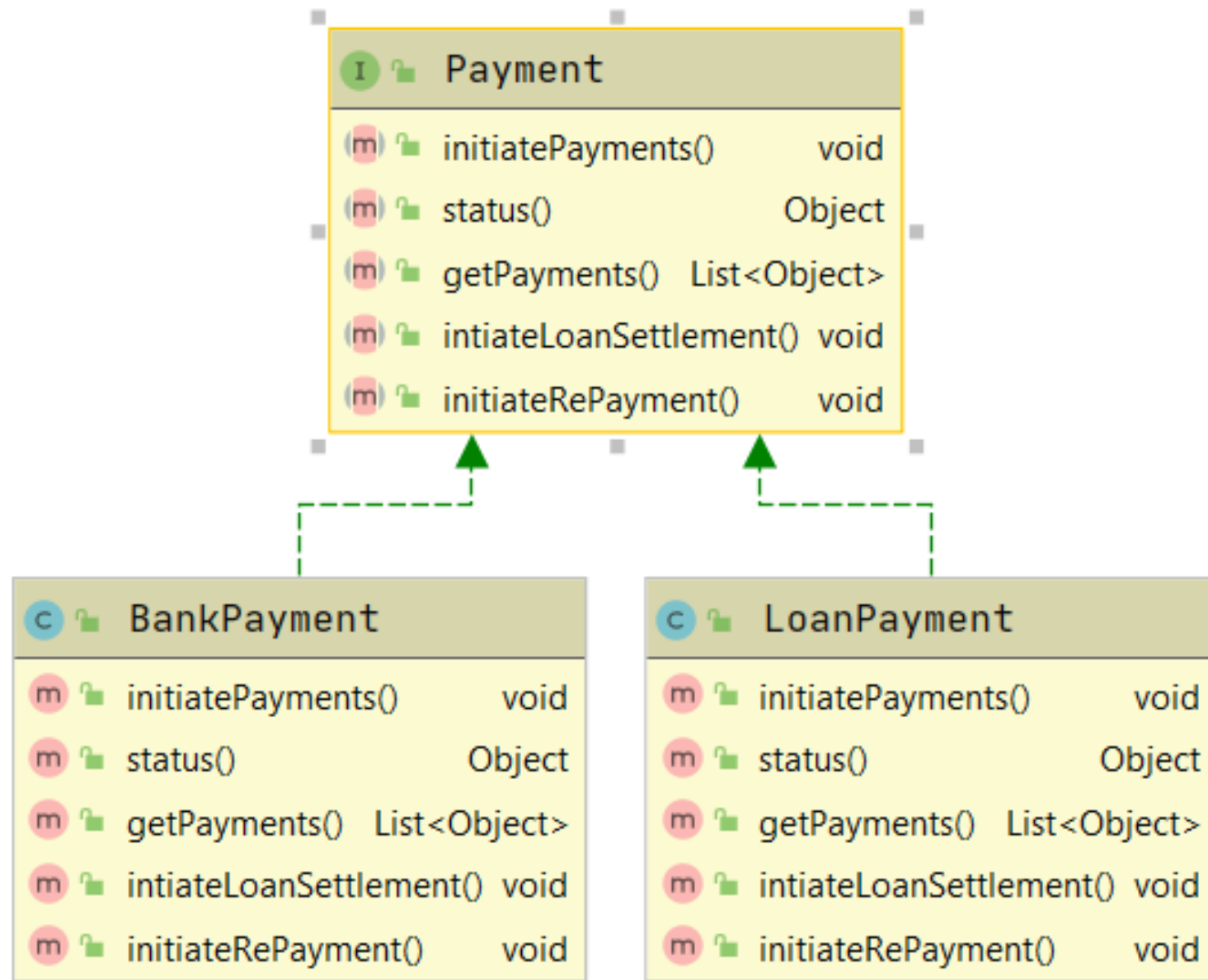


slido



Please analyse the following design and give your comments.

① Start presenting to display the poll results on this slide.



Powered by: Elice


```
public class BankPayment implements Payment {

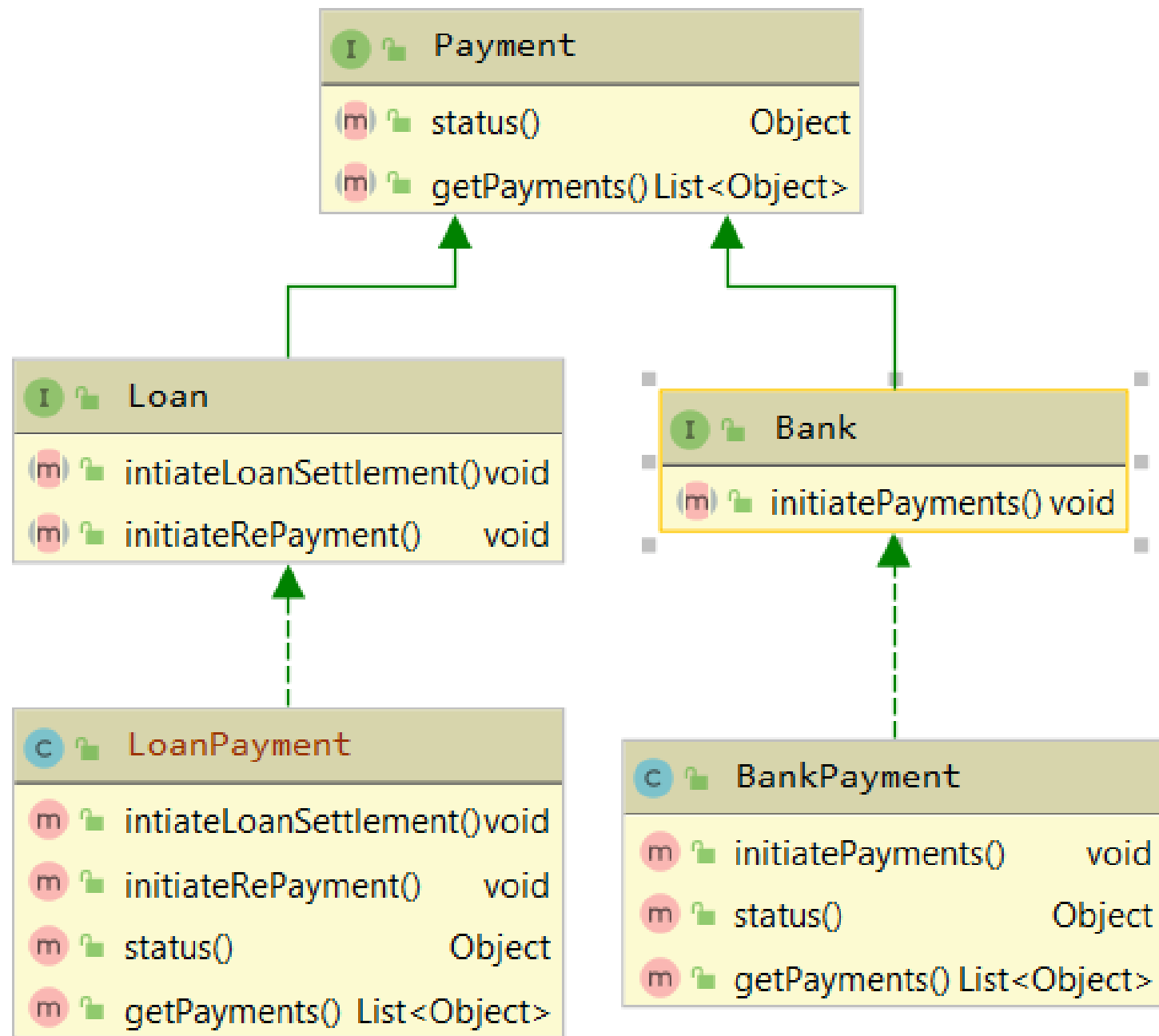
    @Override
    public void initiatePayments() {
        // ...
    }

    @Override
    public Object status() {
        // ...
    }

    @Override
    public List<Object> getPayments() {
        // ...
    }

    @Override
    public void initiateLoanSettlement() {
        throw new UnsupportedOperationException("This is not a loan payment");
    }

    @Override
    public void initiateRePayment() {
        throw new UnsupportedOperationException("This is not a loan payment");
    }
}
```



SOLID Design Principles

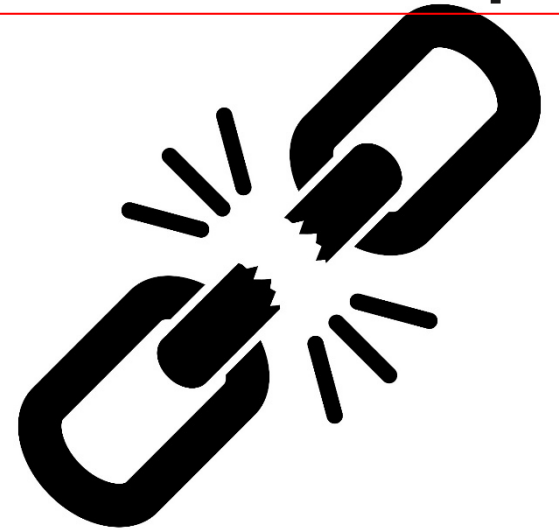
Use Interface/abstract class more

Dependency Injection Principle (DIP)

- A. A HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.***
- B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.***

It is very difficult to write OO code without creating a dependency on something. But we can make it as low as possible.

Dependency Inversion Principle





What does the Dependency Injection Principle (DIP) primarily suggest?

① Start presenting to display the poll results on this slide.

What does the Dependency Injection Principle (DIP) primarily suggest?

High-level modules should depend on low-level modules.

☐ 0%

Low-level modules should depend on high-level modules.

☐ 0%

Both high-level and low-level modules should depend on abstractions. ☒

☒ 0%

Abstractions should depend on

☐ 0%

Abstractions serve as a contract, specifying what functionalities are needed without binding to the specific details of how those functionalities are carried out.



Which of the following has lowest chance to be modified after the system is developed?

Interface has lowest chance to be modified after the system is developed.

“classes should depend on abstraction but not on concretion”.

Flexibility: You can swap out low-level implementations (like changing databases or APIs) without affecting the high-level module.

Maintainability: It promotes loose coupling and modular design, making the code easier to maintain and extend.

Testability: Abstractions make it easier to create mock objects for testing purposes, as you can inject a mock version of the abstraction without relying on actual implementations.

High-Level and Low-Level Modules

- A **high-level module** typically refers to components that focus on **business logic, application flow, or orchestration**. These modules define **what** the system needs to do, without focusing on **how** it should be done in detail.
 - Focuses on the larger structure or functionality of the system and defines the “what”
- A **low-level module** refers to components that are closer to the actual implementation or execution of tasks. These modules define **how** something is done in detail.
 - Handles the specific implementation details and defines the “how”


```
public interface CalculatorOperation {  
    void perform();  
}
```

```
public class Addition implements  
    CalculatorOperation {  
    private double left;  
    private double right;  
    private double result;  
  
    // constructor, getters and setters  
  
    @Override  
    public void perform() {  
        result = left + right;  
    }  
}
```

```
public class Division implements CalculatorOperation {  
    private double left;  
    private double right;  
    private double result;  
  
    // constructor, getters and setters  
    @Override  
    public void perform() {  
        if (right != 0) {  
            result = left / right;  
        }  
    }  
}
```

```
public class Calculator {  
    public void calculate(CalculatorOperation operation) {  
        if (operation == null) {  
            throw new InvalidParameterException("Cannot perform  
operation");  
        }  
        operation.perform();  
    }  
}
```

slido

Please download and install the
Slido app on all computers you use



In the given program, which class is considered the High-Level Module according to the Dependency Inversion Principle (DIP)?

① Start presenting to display the poll results on this slide.

The Calculator class is the high-level module because it defines the overall calculation logic and depends on the abstraction (CalculatorOperation interface) rather than concrete implementations like Addition or Division.

The actual operations are handled by the low-level modules (Addition and Division), which implement the CalculatorOperation interface.

DIP in this Example

- High-Level Module (Calculator):
 - The high-level module does not depend on any specific low-level module. It only knows about the CalculatorOperation interface, which any new operation can implement.
- Low-Level Modules (Addition, Division):
 - These low-level modules depend on the abstraction (i.e., they implement CalculatorOperation). They don't interact directly with the Calculator class, which helps in avoiding tight coupling between the high-level and low-level modules.

Sample

```
class PaymentService {  
    private PaymentProcessor processor;  
  
    public PaymentService(PaymentProcessor  
processor) {  
        this.processor = processor;  
    }  
  
    public void makePayment() {  
        processor.processPayment();  
    }  
}
```

```
class PayPalProcessor implements PaymentProcessor {  
    public void processPayment() {  
        // Payment logic with PayPal  
    }  
}
```

```
interface PaymentProcessor {  
    void processPayment();  
}
```



In the given program, which module is considered the low-level module according to DIP?

① Start presenting to display the poll results on this slide.

Please analyse the following design and give your comments.

```
class FileLogger {  
    public void log(String message) {  
        // Logic to log message to a file  
    }  
}  
  
class OrderProcessor {  
    private FileLogger logger = new FileLogger();  
  
    public void processOrder(Order order) {  
        try {  
            // Order processing logic  
            logger.log("Order processed");  
        } catch (Exception e) {  
            logger.log("Error processing order");  
        }  
    }  
}
```

In this example, OrderProcessor (a high-level module) is directly dependent on FileLogger (a low-level module), violating the DIP.

Applying DIP

To apply the Dependency Inversion Principle, we introduce an abstraction (interface) for logging and depend on this abstraction instead of a concrete class.



Does the OrderProcessor class violate the Dependency Inversion Principle (DIP)?

① Start presenting to display the poll results on this slide.

slido

Please download and install the
Slido app on all computers you use



What modification would help in applying the Dependency Inversion Principle (DIP) in the provided design?

① Start presenting to display the poll results on this slide.

```
interface Logger {  
    void log(String message);  
}
```

Logger is an abstraction (interface) that defines the contract for logging mechanisms.

```
class FileLogger implements Logger {  
    public void log(String message) {  
        // Logic to log message to a file  
    }  
}
```

FileLogger is a low-level module that implements the Logger interface.

```
class OrderProcessor {  
    private Logger logger;  
  
    public OrderProcessor(Logger logger) {  
        this.logger = logger;  
    }  
  
    public void processOrder(Order order) {  
        try {  
            // Order processing logic  
            logger.log("Order processed");  
        } catch (Exception e) {  
            logger.log("Error processing order");  
        }  
    }  
}
```

OrderProcessor is a high-level module that depends on the Logger abstraction, not on the concrete FileLogger implementation.

This design allows changing the logging mechanism (e.g., to log to a database or an external service) without modifying the OrderProcessor class, adhering to the Dependency Inversion Principle.

How OCP and DIP Work Together

In practice, **OCP** and **DIP** often work hand-in-hand in well-designed systems. For example:

- **OCP** allows the addition of new functionalities (like new types of `PaymentProcessor` classes) without modifying the existing system.
- **DIP** ensures that the high-level logic of the system (e.g., `PaymentService`) is not tightly coupled to these specific implementations and only depends on the abstraction (interface).

By applying both principles, you can achieve a more flexible, maintainable, and scalable system:

- **DIP** decouples modules.
- **OCP** ensures you can extend the system with new features without modifying existing classes.

OCP and DIP Together

- Scenario:
- We want to design a PaymentService that can handle different payment methods (e.g., PayPal, Stripe, etc.) and allow easy extension (adding new payment methods) without modifying the existing code. Additionally, the PaymentService should not depend directly on any specific payment processor to avoid tight coupling.

Sample

```
class PaymentService {  
    private PaymentProcessor processor;  
  
    public PaymentService(PaymentProcessor  
processor) {  
        this.processor = processor;  
    }  
  
    public void makePayment() {  
        processor.processPayment();  
    }  
}
```

```
class PayPalProcessor implements PaymentProcessor {  
    public void processPayment() {  
        // Payment logic with PayPal  
    }  
}
```

```
interface PaymentProcessor {  
    void processPayment();  
}
```

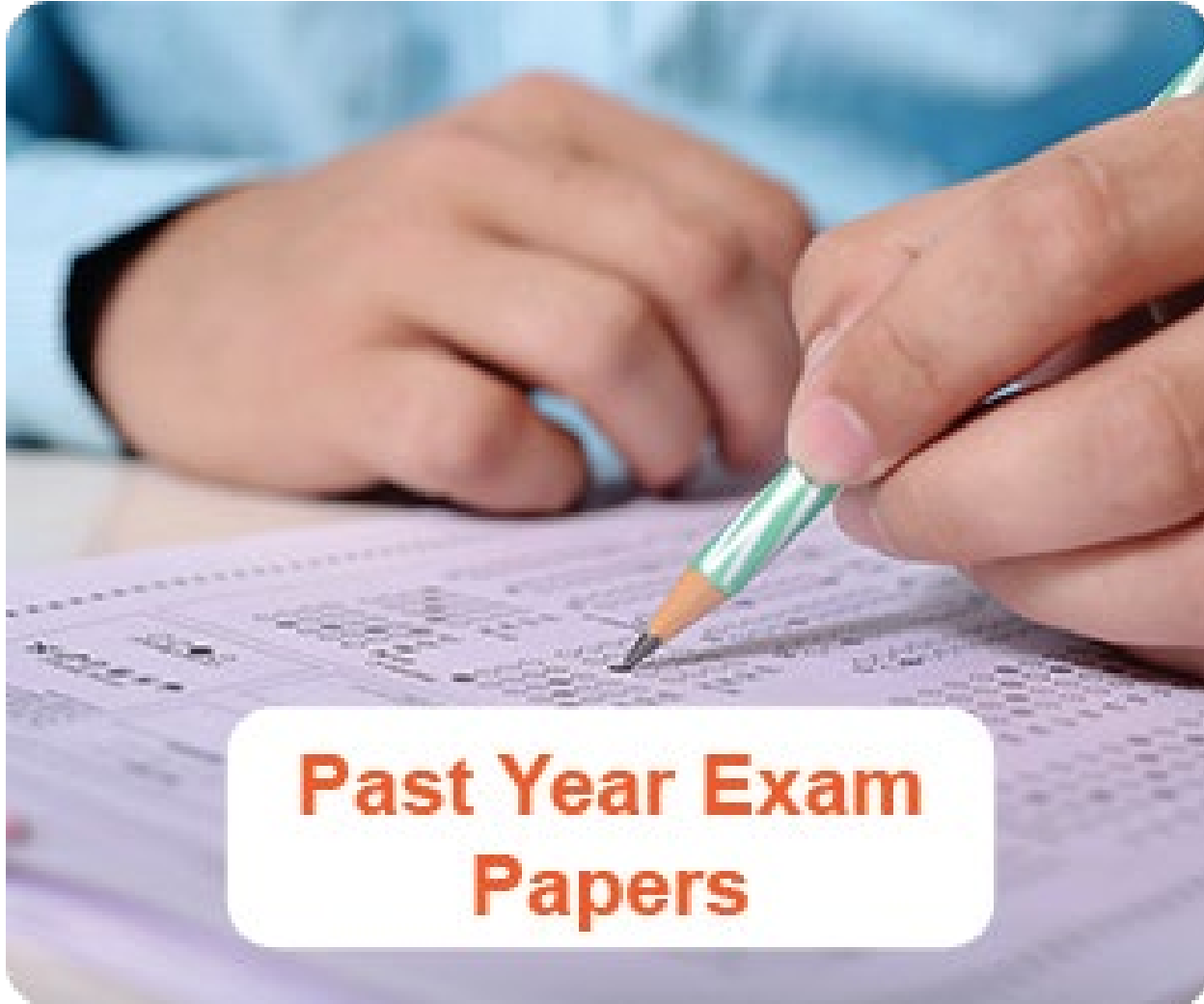
- **OCP (Open-Closed Principle):**

- We can extend the system by adding new `PaymentProcessor` implementations (e.g., `BitcoinProcessor`, `StripeProcessor`) without modifying existing classes like `PaymentService` or `PayPalProcessor`.

- **DIP (Dependency Inversion Principle):**

- The `PaymentService` class, which is a high-level module, does not depend on any concrete payment processor (e.g., `PayPalProcessor`). Instead, it depends on the `PaymentProcessor` interface, which is an abstraction, making the system more flexible and loosely coupled.

PYP revision



2017-18 S1

4(b) The UML **Class Diagram** in Appendix E shows the relationship between FOUR classes and snippet code of some of the methods implementation. Both `SavingAccount` and `InvestmentAccount` classes have a reference to the `DataPresenter` object in order to know the type of format (XML or HTML). `DataPresenter` class also has a reference each to the `SavingAccount` and `InvestmentAccount` objects in order to read their content and present the format accordingly.

It is suggested that a new `RetirementAccount` class will be added and the data should be presented also in JSON format.

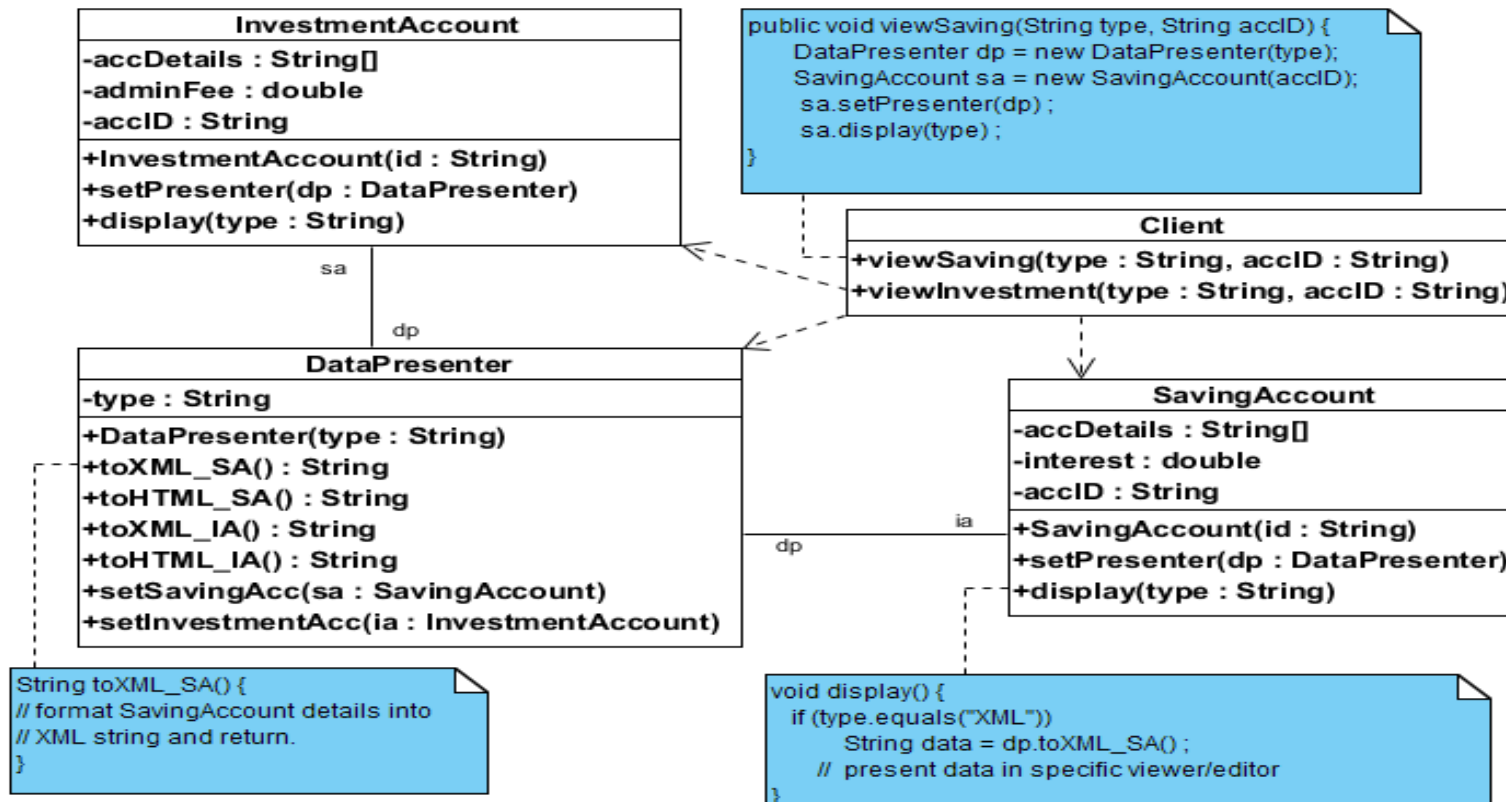
(i) State and explain ONE design issue of the current design.

(3 marks)

(i) Suggest and explain, with a Class Diagram, how you can improve the current design to cater to the new requirements in your design with reusability, extensibility and maintainability in mind. State the design principle(s) you have applied. *You should show the class/interface method(s) to illustrate your idea.*

(7 marks)

4(b) The UML **Class Diagram** in Appendix E shows the relationship between FOUR classes and snippet code of some of the methods implementation. Both `SavingAccount` and `InvestmentAccount` classes have a reference to the `DataPresenter` object in order to know the type of format (XML or HTML). `DataPresenter` class also has a reference each to the `SavingAccount` and `InvestmentAccount` objects in order to read their content and present the format accordingly.





**What design issue is present
in the current DataPresenter
class?**

① Start presenting to display the poll results on this slide.

slido

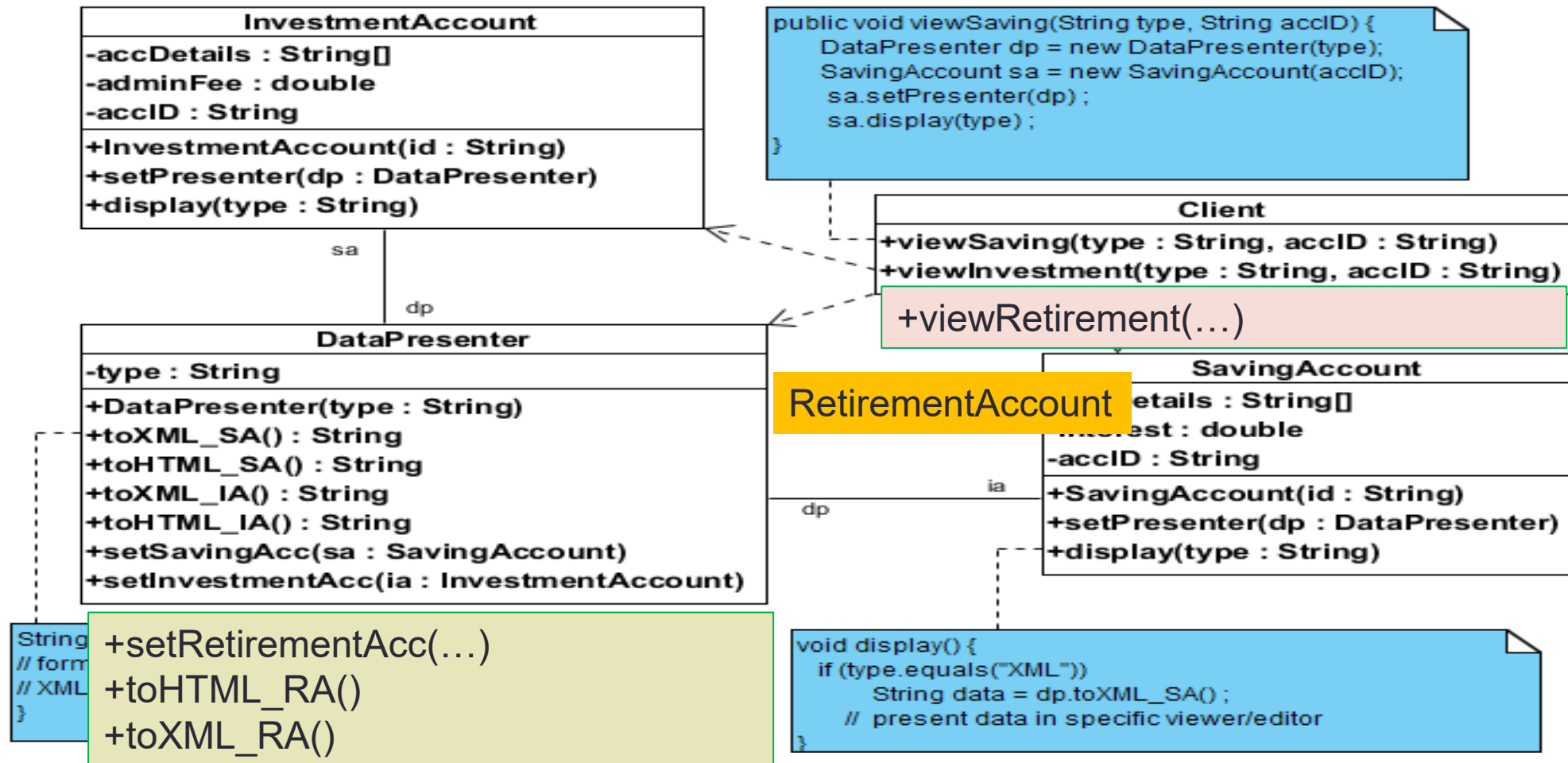
Please download and install the
Slido app on all computers you use



Which classes require modification to incorporate the RetirementAccount class into the existing design?

① Start presenting to display the poll results on this slide.

It is suggested that a new RetirementAccount class will be added



Violates OCP: cannot extend easily. Adding new account type has big impact of change

slido

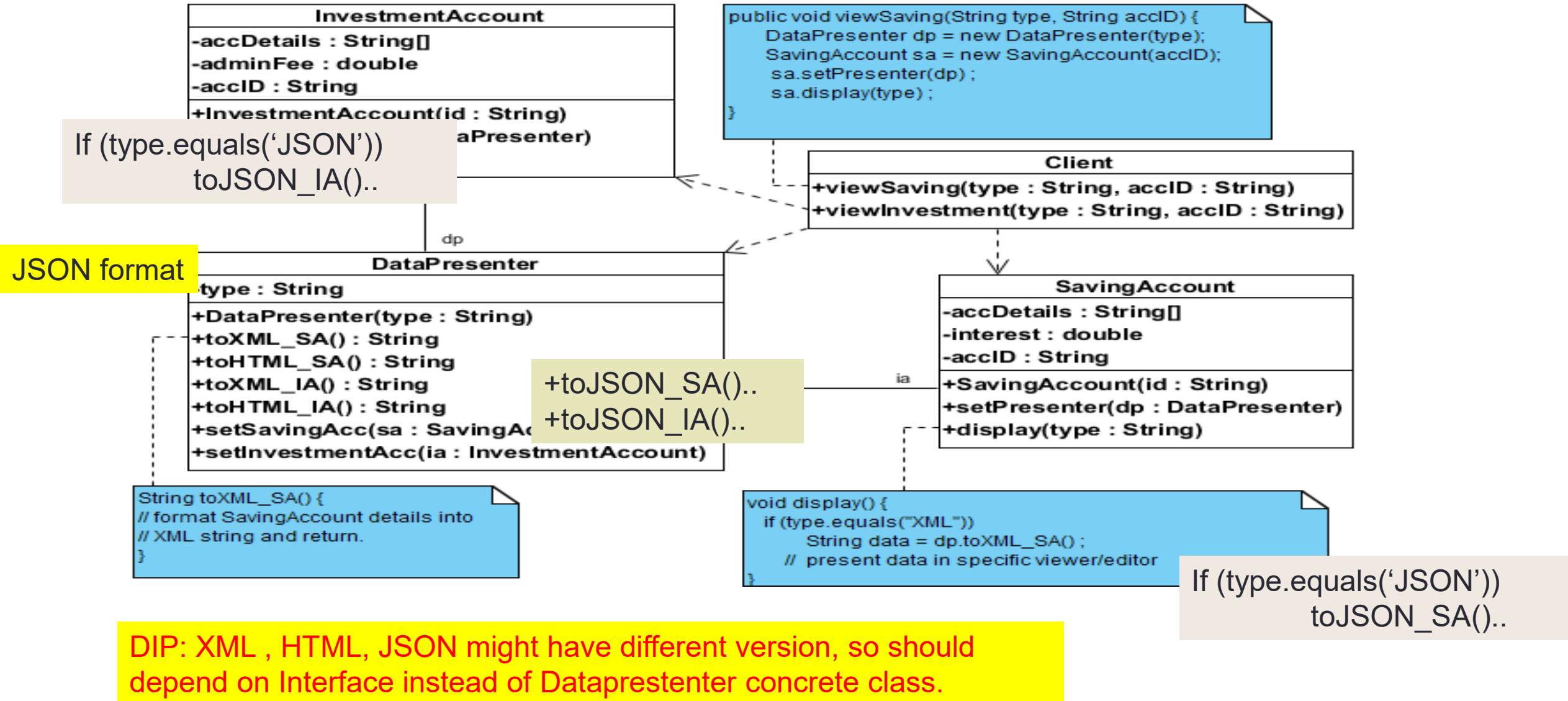
Please download and install the
Slido app on all computers you use



Which of the following classes must be modified to add JSON format support to the current design?

① Start presenting to display the poll results on this slide.

It is suggested that the data should be presented also in JSON format.



4bi Q:State and explain ONE design issue of the current design.

(3 marks)

Ans: 4b (i) (3 marks)

Any of Bad encapsulation, Tight coupling and Low Cohesion

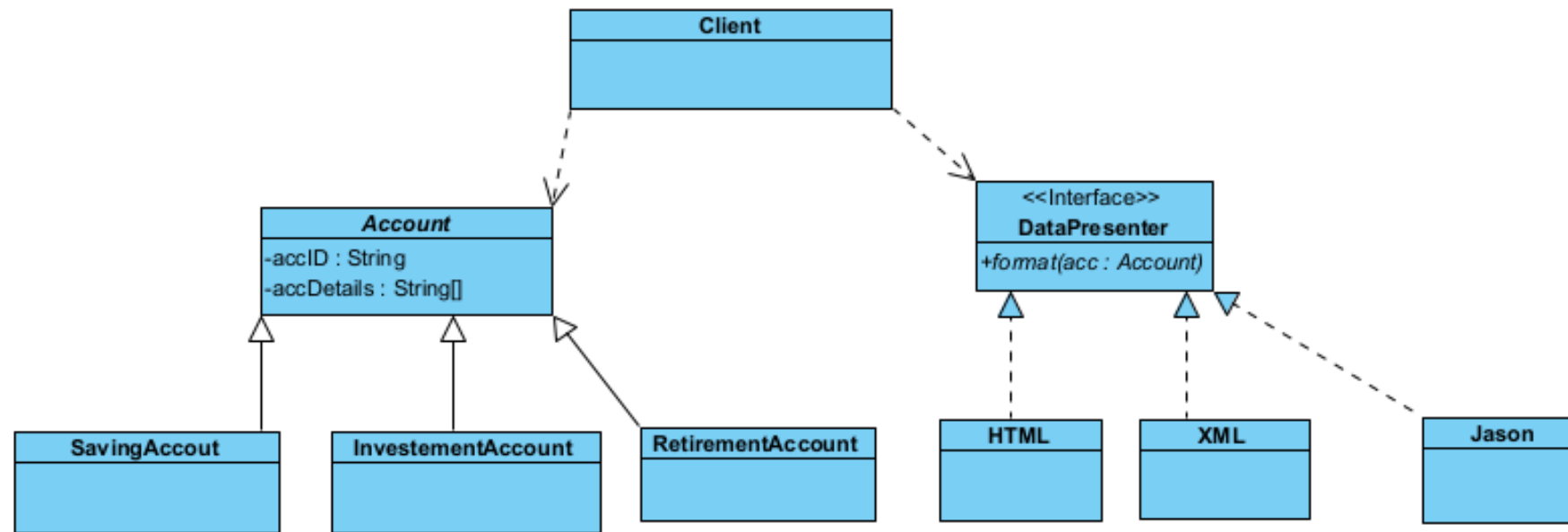
OCP: cannot extend easily. Adding new account type has big impact of change

DIP: XML , HTML, JSON might have different version, so should depend on Interface instead of Dataprestenter concrete class.

SRP: Dataprestenter has more than one reasons to be changed

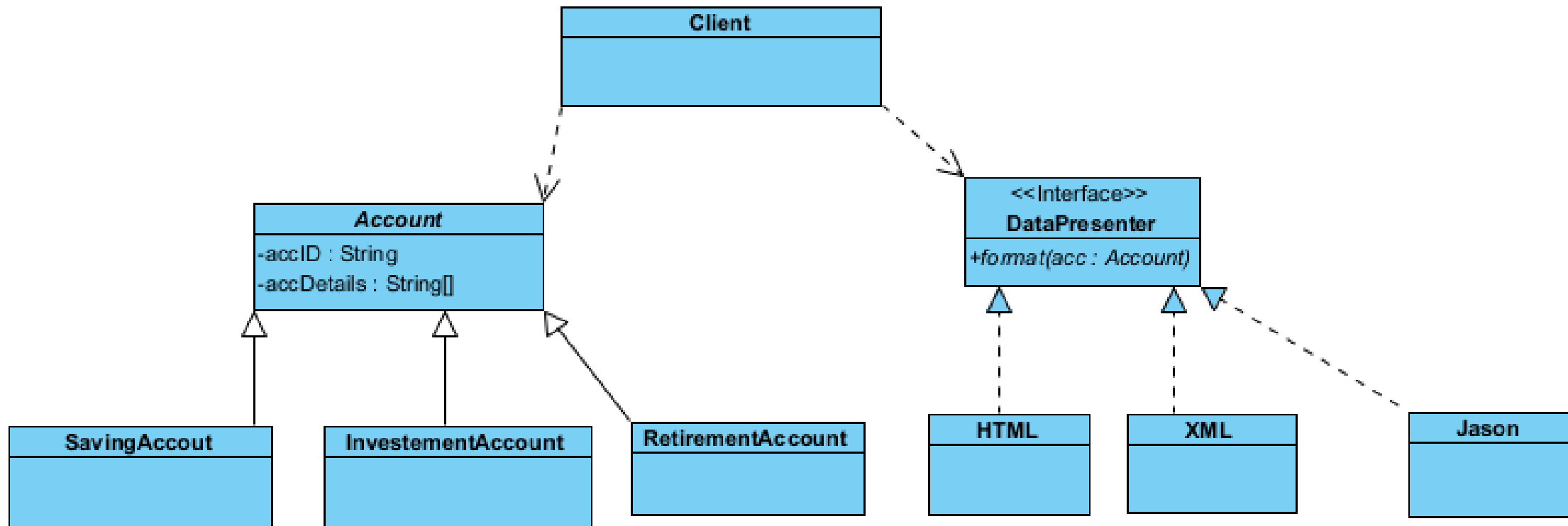
4bii Q: Suggest and explain, with a Class Diagram, how you can improve the current design to cater to the new requirements in your design with reusability, extensibility and maintainability in mind. State the design principle(s) you have applied. *You should show the class/interface method(s) to illustrate your idea.*

(7 marks)



4 marks

- have an Account abstract/concrete class
- each saving, investment and retirement account are subclasses
- have a DataPresenter interface with a format(Account)/display(..) method
- each XML, HTML, JSON classes will override format(Account)/display(..) method
- *explain the benefit of client class*



explain using either one of the design principles. (3 marks)