SOFTWARE DESIGN & ANALYSIS (Week-14)

USAMA MUSHARAF

MS-CS (Software Engineering)

LECTURER (Department of Computer Science)

FAST-NUCES PESHAWAR

SOLID Principle

1. Single Responsibility Principle

2. Open Close Principle

3. Liskov Substitution Principle

4. Interface Segregation Principle

5. Dependency Inversion Principle

SINGLE RESPONSIBILITY PRINCIPLE

SINGLE RESPONSIBILITY PRINCIPLE

- Single responsibility means that your class (any entity for that matter, including a method in a class, or a function in structured programming) should only do one thing.

- A class should have only one reason to change.

- Related to Coupling & Cohesion.

SINGLE RESPONSIBILITY PRINCIPLE

What a class does?

- The more a class does, the more likely it will change.
- The more a class changes, the more likely we will introduced bugs.

One of the simples principle but one of the most difficult to get right.

SRP EXAMPLE

```
Class Post{
void CreatePost (Database db, string postMessage) {
try{
    db.Add(postMessage);
    }
catch (Exception ex)
    {
    db.LogError("An error occurred:", ex.ToString());
    File.WriteAllText("\LocalErrors.txt", ex.ToString());
    }
  }
}
```

SOLUTION

```
Class Post{

Private ErrorLogger errorlLogger = new ErrorLogger();

void CreatePost (Database db, string postMessage) {

try{

    db.Add(postMessage);

    }

catch (Exception ex)

    {

    errorLogger.log(ex.ToString());

    }

  }

}
```

```
Class ErrorLogger {

    void log(string error)

    {

    db.LogError("An error occurred:", error);

    File.WriteAllText("\LocalErrors.txt",
error);

        }

}
```

OPEN CLOSE DESIGN PRINCIPLE

OPEN CLOSE DESIGN PRINCIPLE

- "Software entities like classes, modules and functions should be open for extension but closed for modifications"

Bertrand Meyer

OPEN CLOSE DESIGN PRINCIPLE

- The **_Open Close Principle_** states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code.

- The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

OPEN CLOSE DESIGN PRINCIPLE

Open-closed principle have two primary attributes:

- I. They are "Open For Extension"

  This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.

- II. They are "Closed for Modification"

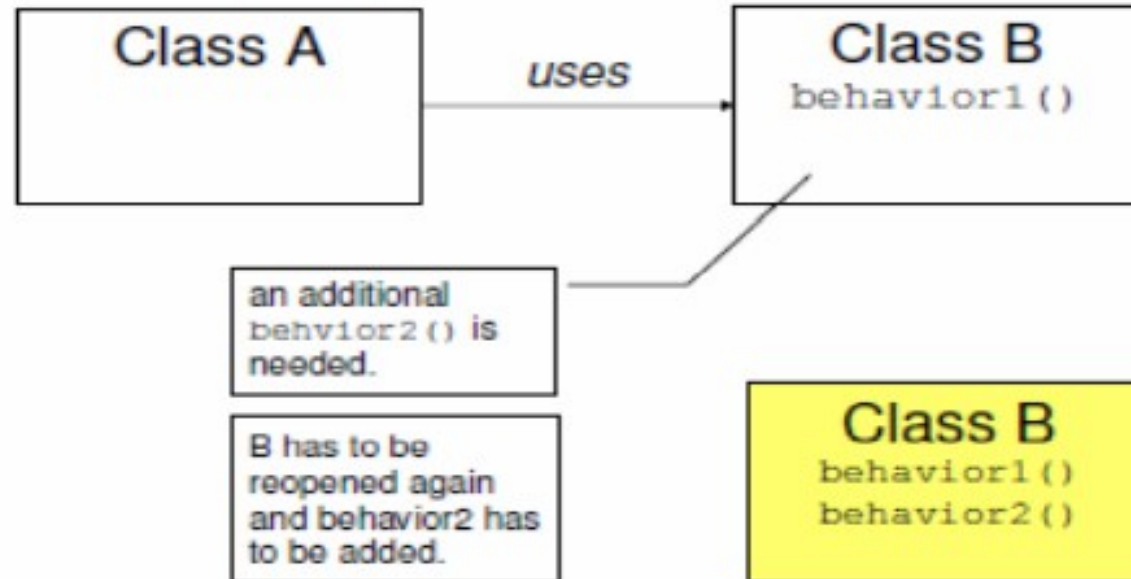 No one is allowed to make changes in source code.

OPEN CLOSE DESIGN PRINCIPLE

- The Open/Closed principle can be applied in be applied in object oriented paradigms with the help of inheritance and polymorphism:

OPEN CLOSE DESIGN PRINCIPLE

OPEN CLOSE DESIGN PRINCIPLE

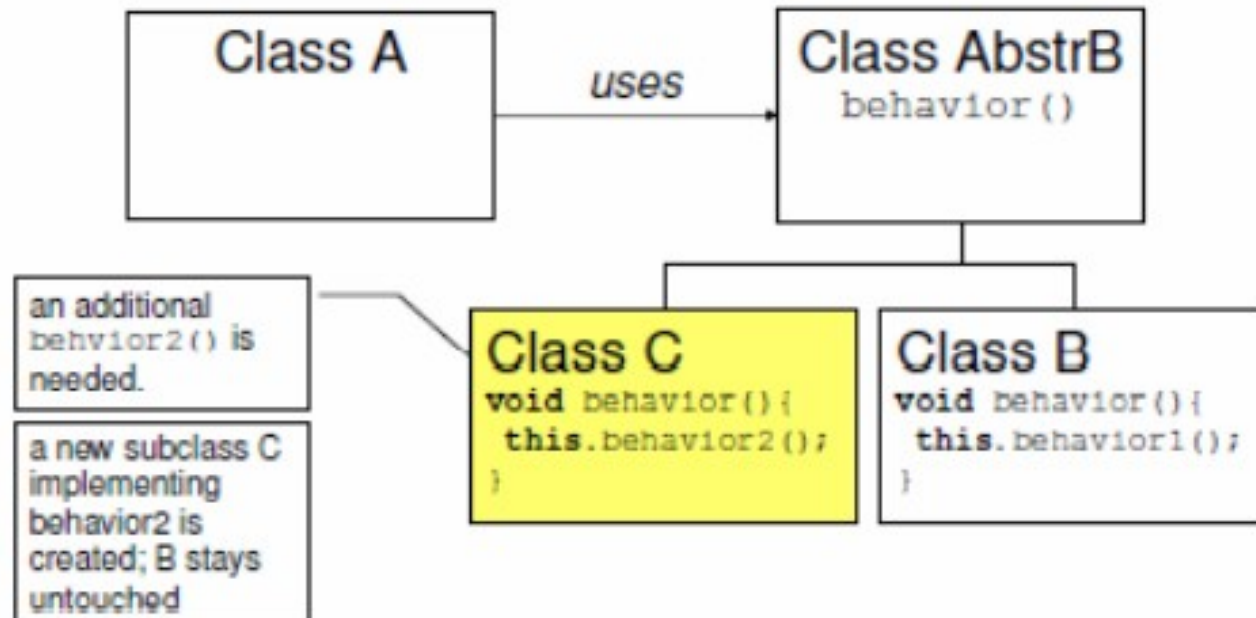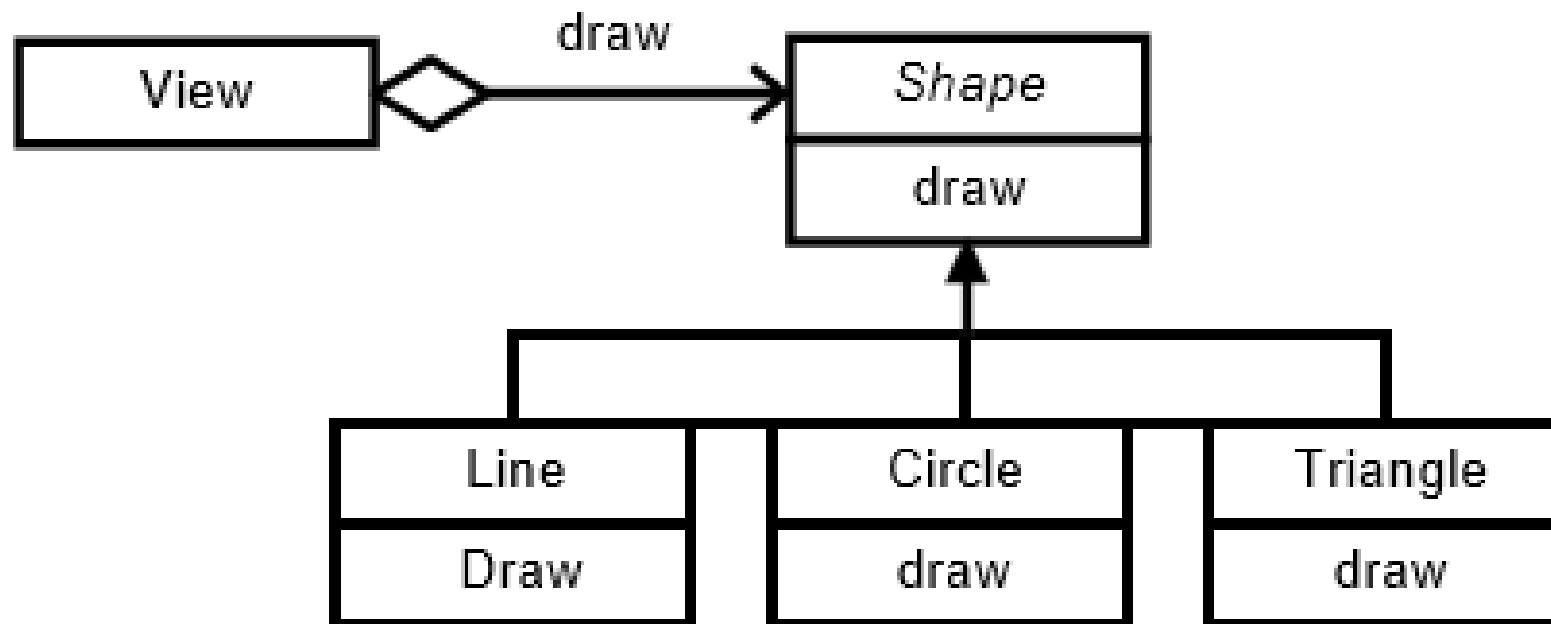The "Open/Closed principle" – Example

After (with open/closed)

Class A — *uses* → Class AbstrB
behavior()

an additional behvior2() is needed.

a new subclass C implementing behavior2 is created; B stays untouched

Class C
void behavior(){
  this.behavior2();
}

Class B
void behavior(){
  this.behavior1();
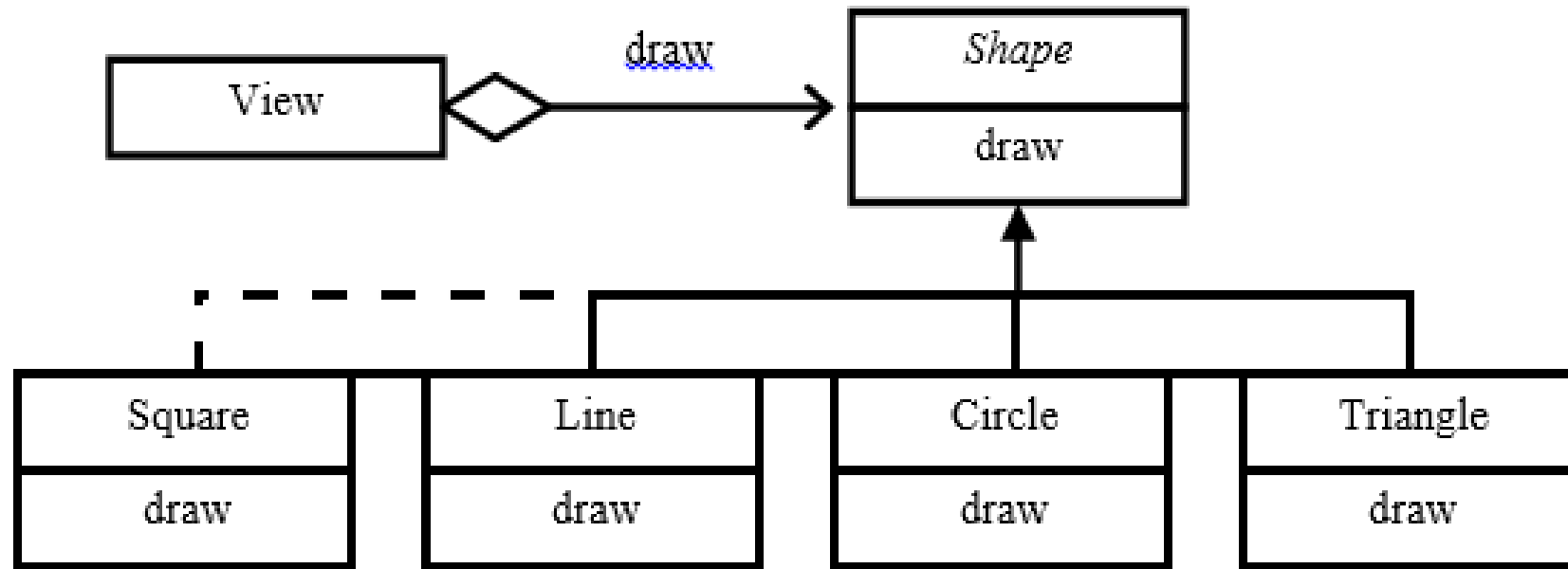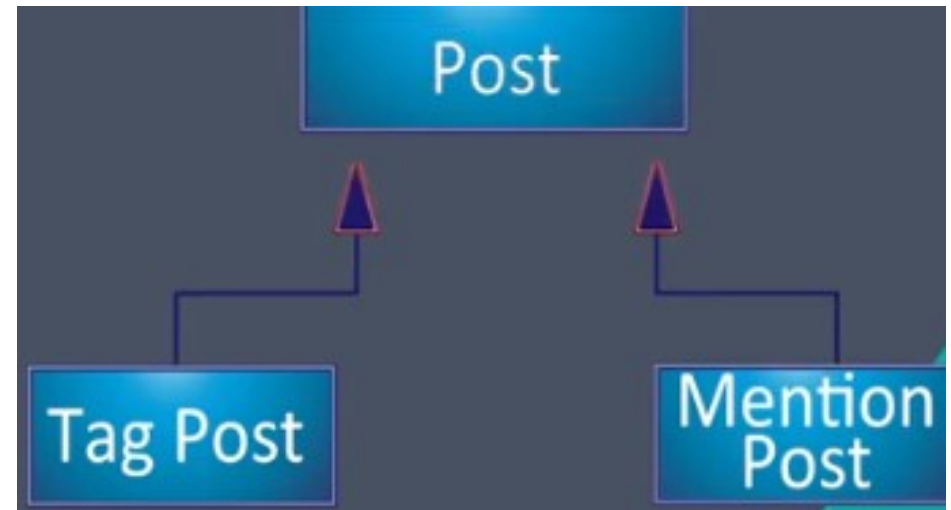}

EXAMPLE

EXAMPLE

In general, polymorphism is a powerful tool to develop flexible and reusable systems

OCP EXAMPLE

```
Class Post{
void CreatePost (Database db, string postMessage) {
    If (postMessage.StartsWith("#"))
    {
    db.AddAsTag(postMessage);
    }
    else
    {
    db.Add(postMessage);
    }
      }
    }
```

SOLUTION

```
class Post
{
    void CreatePost(Database db,
                            string postMessage)
    {
        db.Add(postMessage);
    }
}
```

LISKOV SUBSTITUTION PRINCIPLE

LISKOV SUBSTITUTION PRINCIPLE

- The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass.

- *If substituting a superclass object with a subclass object changes the program behavior in unexpected ways, the LSP is violated.*

- *The LSP is applicable when there's a super-type sub-type inheritance relationship by either extending a class or implementing an interface.*
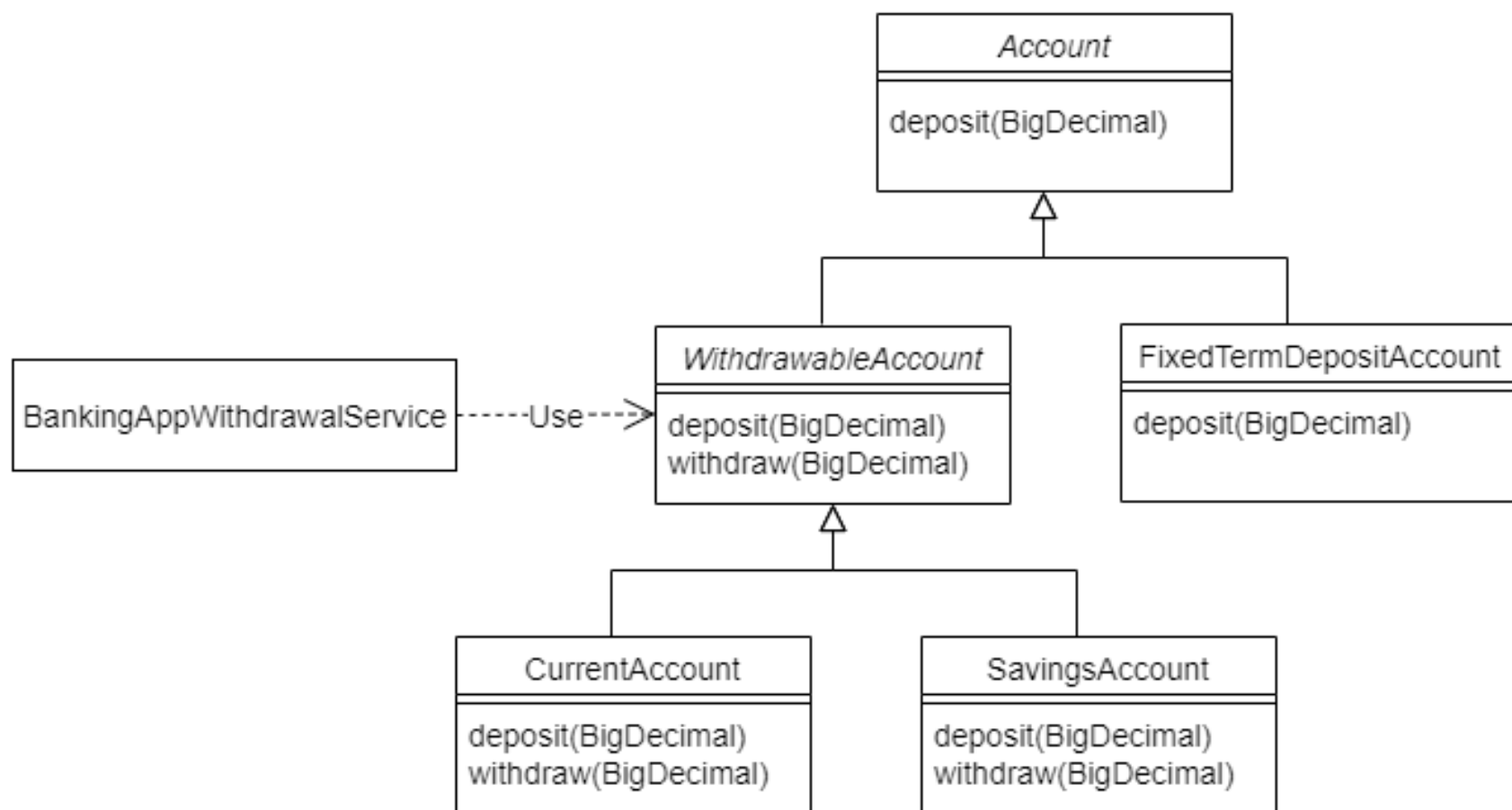
LSP IMPORTANCE

- *Liskov Substitution Principle actually teaches the exact meaning of inheritance.*

EXAMPLE
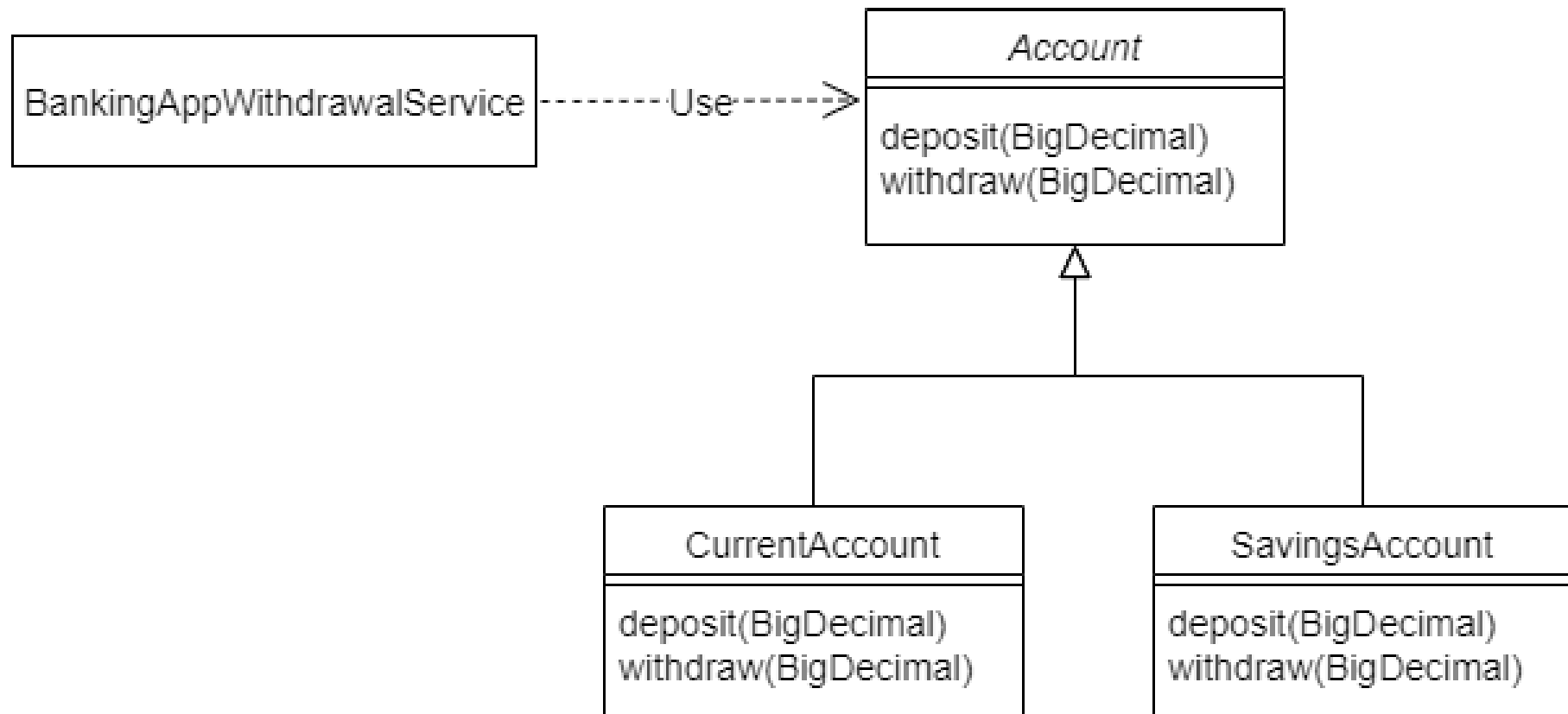
EXAMPLE

INTERFACE SEGREGATION PRINCIPLE

INTERFACE SEGREGATION PRINCIPLE

- *"Clients should not be forced to depend upon interfaces that they do not use."*

INTERFACE SEGREGATION PRINCIPLE

- If you have an abstract class or an interface, then the implementers should not be forced to implement parts that they don't care about.

- *In programming, the ISP states that no client should be forced to depend on methods it does not use.*

INTERFACE SEGREGATION PRINCIPLE

- *Don't add additional functionality to an existing interface by adding new methods.*

- *Instead create a new interface and let your class implement multiple interfaces if needed.*

INTERFACE SEGREGATION PRINCIPLE

- ISP deals with non cohesive interfaces and it reduces coupling in a system.

ISP (EXAMPLE)

Interface iPost {

Void CreatPost();

Void ReadPost();

}

Violation of ISP

Interface iPostCreate {

Void CreatPost();

}

Interface iPostRead {

Void ReadPost();

}

ISP (EXAMPLE)

Violation of ISP

Interface ISmartDevice{

void Print();

void Fax();

void Scan();

}

class AllinOnePrinter implements ISmartDevice{

public void Print() {

}

public void Fax() {

}

public void Scan() {

}

}

ISP EXAMPLE

Now suppose we need to handle a new device (EconomicPrinter class) that can only print. We're forced to implement the Whole interface.

```
class EconomicPrinter implements
Iprinter{
    public void Print()   {
        }
    public void Fax()   {
    throw new NotSupportedException();
        }
    public void Scan() {
    throw new NotSupportedException();
        }
}
```

SOLUTION

interface Iprinter {

void Print();

}

interface Ifax {

void Fax();

}

interface Iscanner{

void Scan();

}

class   EconomicPrinter   implements ISmartDevice{

   public void Print()   {

         }

      }

SOLUTION

```
class EconomicPrinter implements
Iprinter, Ifax, Iscanner{
    public void Print()   {

        }
    public void Fax()   {

        }
    public void Scan() {

        }

    }
```

ISP

- The ISP guides us to create many small interfaces with coherent functionalities instead of a few big interfaces with lots of different methods.

- When we apply the ISP, class and their dependencies communicate using focused interfaces, minimizing dependencies.

- Smaller interfaces are easier to implement, improving flexibility and the possibility of reuse.

DEPENDENCY INVERSION PRINCIPLE

DEPENDENCY INVERSION PRINCIPLE

1. High Level Modules should not depend on low level modules. Both should depend on abstractions.

2. Abstractions should not depend upon details. Details should depend upon abstractions.

DEPENDENCY INVERSION PRINCIPLE

- In programming., the dependency inversion principle is a way to decouple software modules.

- OCP, LSP, and even SRP lead to the dependency inversion principle.

DEPENDENCY INJECTION

Dependency Injection

- Injecting any dependencies of a class through a class constructor as an input parameter.

- Injection basically converts composition to association.

- Strong coupling to weak coupling.

DIP (EXAMPLE)

```
Class Post{

Private ErrorLogger errorlLogger = new ErrorLogger();

void CreatePost (Database db, string postMessage) {

try{

    db.Add(postMessage);

    }

catch (Exception ex)

    {

    errorLogger.log(ex.ToString());

    }

  }

}
```

Dependency

```
Class ErrorLogger {

    void log(string error)

    {

    db.LogError("An error occurred:", error);

    File.WriteAllText("\LocalErrors.txt", error);

    }

}
```

SOLUTION

Class Post{

Private Logger _logger;

Public Post (Logger injectedLogger){

    logger = injectedLogger;

    }

void CreatePost (Database db, string postMessage)
{

    {

    .....

    .....

    }

}

Dependency Injection

HAVE A GOOD DAY!