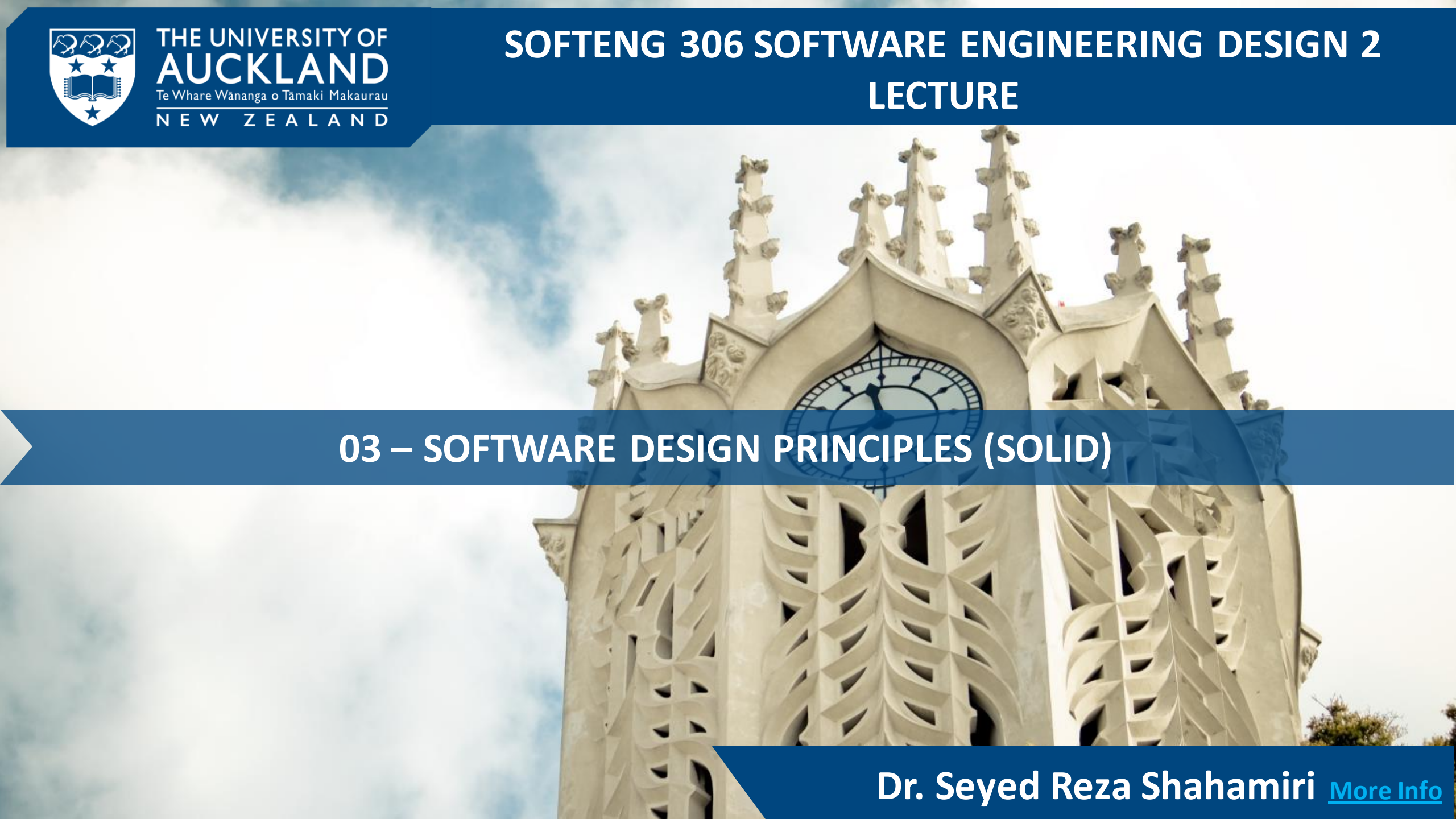


THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SOFTENG 306 SOFTWARE ENGINEERING DESIGN 2 LECTURE

03 – SOFTWARE DESIGN PRINCIPLES (SOLID)

Dr. Seyed Reza Shahamiri More Info

SOLID Design Patterns

- SOLID is an acronym for a set of software design principles to make software designs more:
  - Understandable,
  - Flexible,
  - Maintainable.

- They were introduced by different people, but initially promoted by Robert C. Martin as a set of principles.

Single Responsibility Principle (SRP)

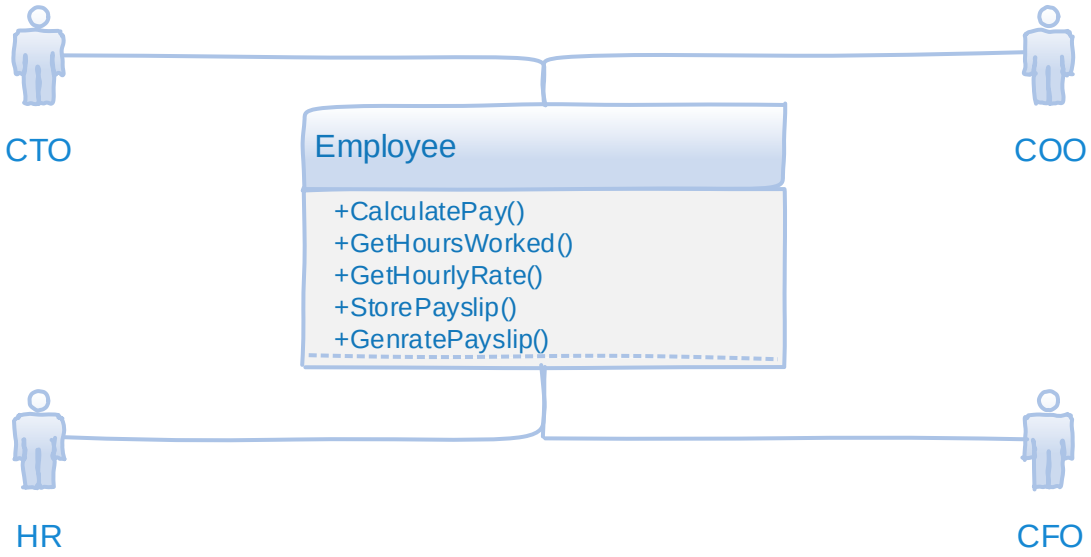
"A class should have only one reason to change."

- This principle recommends that
  1. A method should only perform one task, and
  2. A class should be responsible to one actor.

Single Responsibility Principle (SRP)

Example

Suppose an Employee class with the following operations:

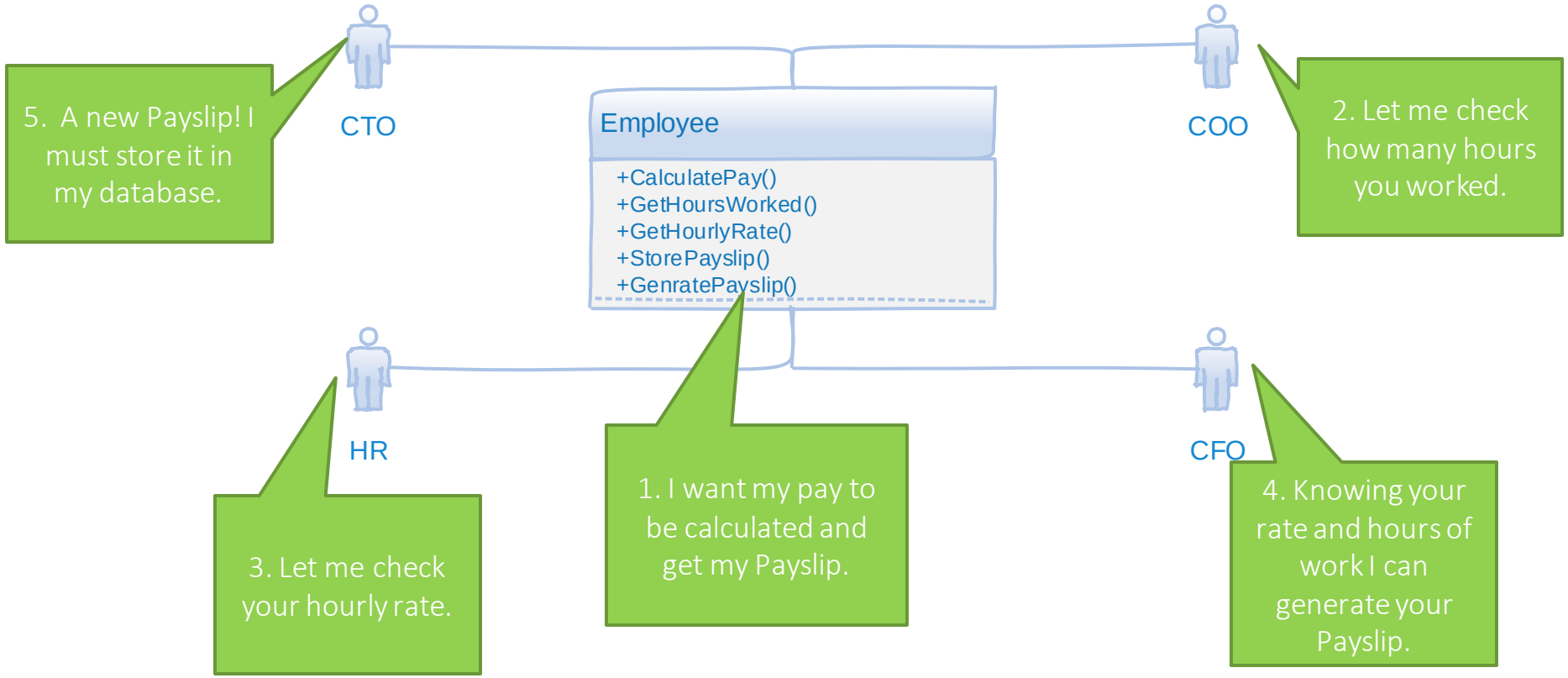


CTO: Chief Technology Officer
COO: Chief Operations Officer
CFO: Chief Finance Officer
HR: Human Resources

Single Responsibility Principle (SRP)

Example

Do you smell anything? (Rigidity? Fragility? Immobility? Viscosity ? Needless Complexity? Needless Repetition? Opacity?)

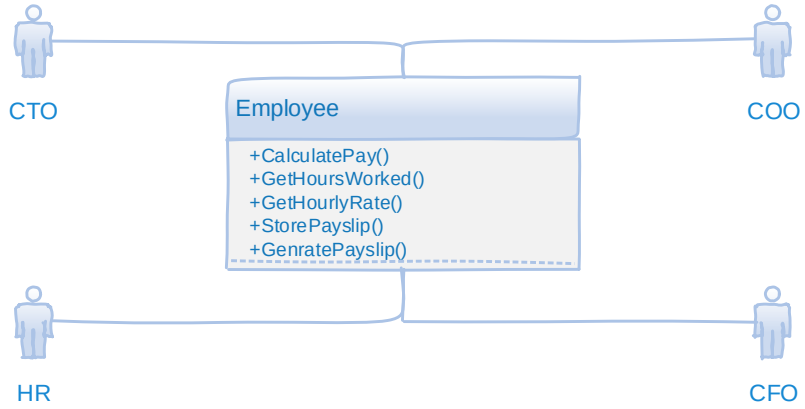


Single Responsibility Principle (SRP)

Example

Problem

- This design enforces all these four actors to be tightly coupled via Employee class.
- Now, what happens if HR decides to change the way hours are calculated?
  - Then Employee class must change, that may lead to affecting all actors and their departments!
- Or, what happens if HR supplies wrong hours?
  - Then the organization pays too much salary, and COO looses on her budget!
  - In this case, whom the CEO should blame?
- This is because all these actors have interests in Employee class and side effects effecting them all.



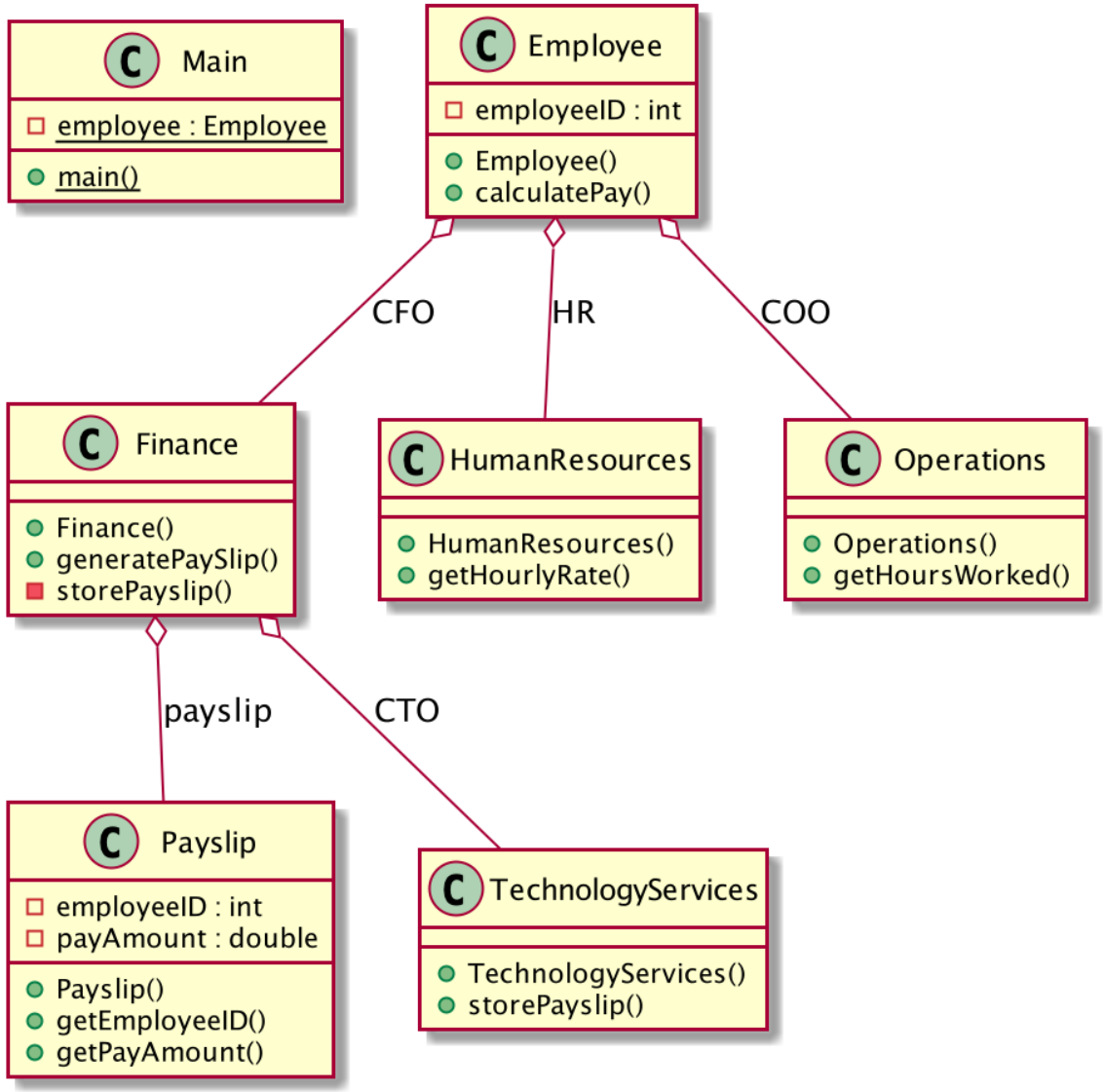
Single Responsibility Principle (SRP)

Example

Partial Solution

- A partial solution is to ask the actors to take responsibility of the methods related to them and implement them themselves.
  - That means to add separate classes for Finance, Operations, HR, etc. and add to them operations that are their responsibility.
- SRP tells us to separate methods in different classes, if they change for different reasons.

That's Encapsulation and Information Hiding



Single Responsibility Principle (SRP)

Example

Full Solution using Interfaces

- A better solution is to minimize this coupling and side effects by creating additional interfaces for classes of these actors so that Employee class deals with these interfaces only.
- The interfaces take responsibilities of interacting with Employee class and hide the implementation details of these methods.
- What is an interface?

That's even more Information Hiding!

Single Responsibility Principle (SRP)

Interfaces

- An interface is an abstraction and encapsulation mechanism that enables a class to advertise its public operations to other classes.
- An interface simply provides information about inputs and outputs to class public methods without any implementation details.
- The implementations of the public methods are still handled by the class.

Single Responsibility Principle (SRP)

Interfaces

Analogy

- An analogy for interfaces is a TV and its remote control:
  - The remote control is an interface to TV operations. It has buttons that enable users to operate the TV.
  - But the actual operations are implemented in the TV.
  - As a user, you don't care which operations are involved when you press the power button to turn on the TV.
  - TV manufactures can change all these operations without any need to change the remote control, that means as the user's point of view, nothing is changed.

Single Responsibility Principle (SRP)

Interfaces

- Using an interface, other objects deal with the interface instead of the class directly.
- This way any future changes to the class has no side effect on the using objects unless the changes are so significant that the interface is also needs to be changed.
- Nevertheless, most of the times the method signatures are not changed but their operations are changed. Hence, no interface change is required.

Single Responsibility Principle (SRP)

Interfaces

Java Interfaces

- A class that implements an interface must provide **public** implementations for all interface methods.
- The class must also follow the **signature** of the interface methods precisely.
- A class can have more methods/properties/attributes as well (usually private).
  - If so, these additional methods won't be accessible by the interface.

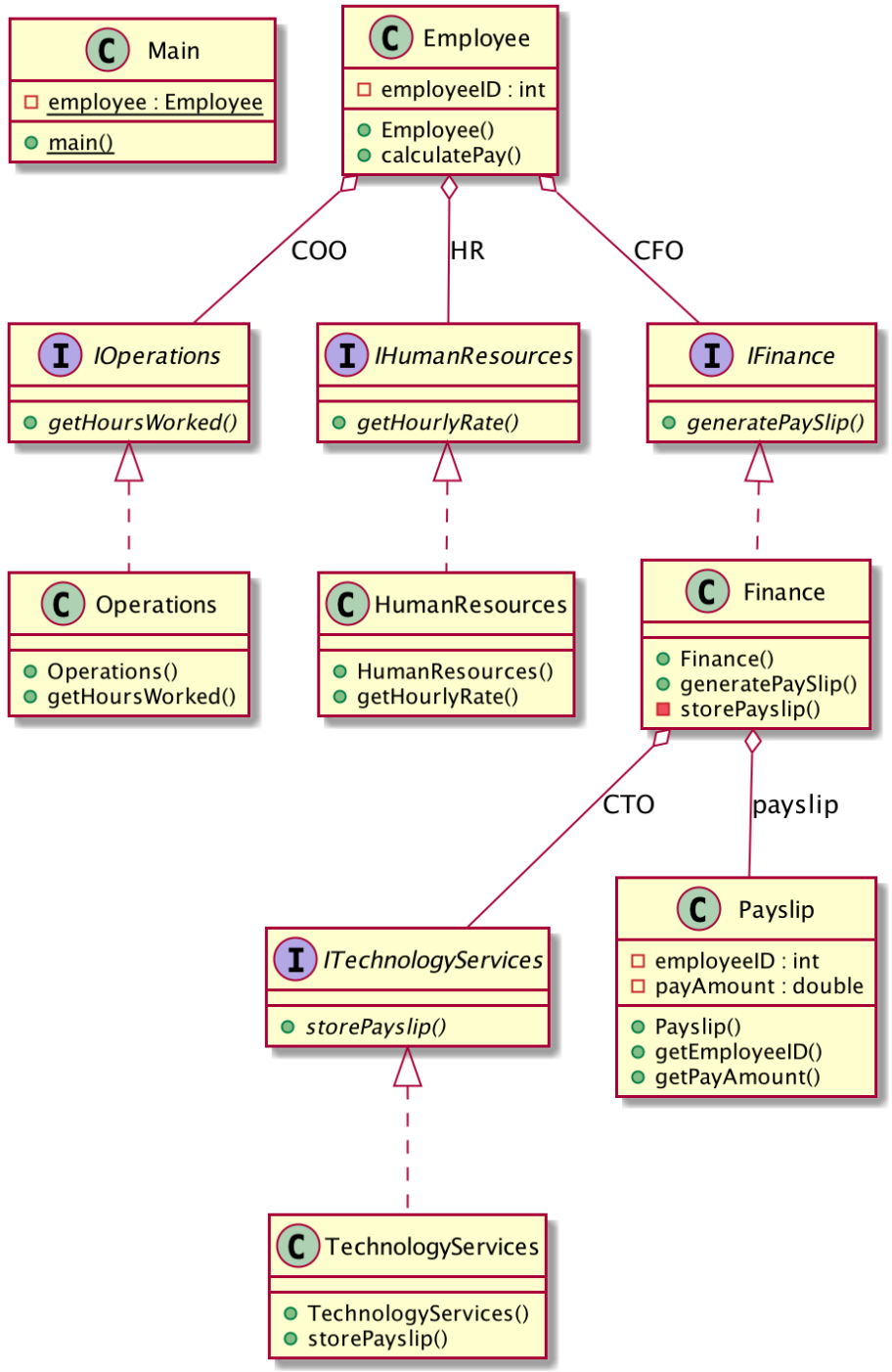
Usage: declare object of the interface, instantiate the object using one of the classes that implements the interface.

```
interface IMyClass
{
    void operation1 (int anArgument);
    int operation2();
}
public class MyClass implements IMyClass{

    public void operation1 (int anArgument)
    {
        // Implementation of operation 1
    }

    public int operation2()
    {
        // Implementation of operation 2
        return 0;
    }
}

public class UserOfMyClass
{
    public void useInterface()
    {
        IMyClass myclass = new MyClass();
    }
}
```

Single Responsibility Principle (SRP)

Example

A High Level Solution



Single Responsibility Principle (SRP)

- SRP extends to architectural components as well, when a component is composed of multiple classes:
  - That is, if multiple classes change for the same reason, they belong to the same component (aka package), and if classes change for different reasons, they should be in different components.
  - This component principle is called the Common Closure Principle (CCP).

Open-Close Principle (OCP)

"Software entities should be open for extension, but closed for modification."

- We expect class (or other entities) behavior to change overtime.
- Nevertheless, we must avoid any side effects of these change to other coupled classes because ay modification to a class may lead to modifications to other classes.

Open-Close Principle (OCP)

- To do so, we allow any modifications to the class as long as no change is broadcasted to other entities.
- Otherwise, the architecture should be designed in a way that allows easy extractions via inheritance, abstractions, and interfaces, so that these additional functionalities and changes can be added easily to new classes derived from the original interface!
- Proper usage of interfaces allows seamless OCP integration (see Dependency Inversion Principle for more details)

Open-Close Principle (OCP)

Which design smell(s) does OCP prevent?

Rigidity

The system is hard to change because every change forces many other changes to other parts of the system.

Fragility

Changes cause the system to break in places that have no conceptual relationship to the part that was changed.

Immobility

It is hard to disentangle the system into components that can be reused in other systems.

Viscosity

Doing things right is harder than doing things wrong

Needless Complexity

The design contains infrastructure that adds no direct benefit.

Needless Repetition

The design contains repeating structures that could be unified under a single abstraction.

Opacity

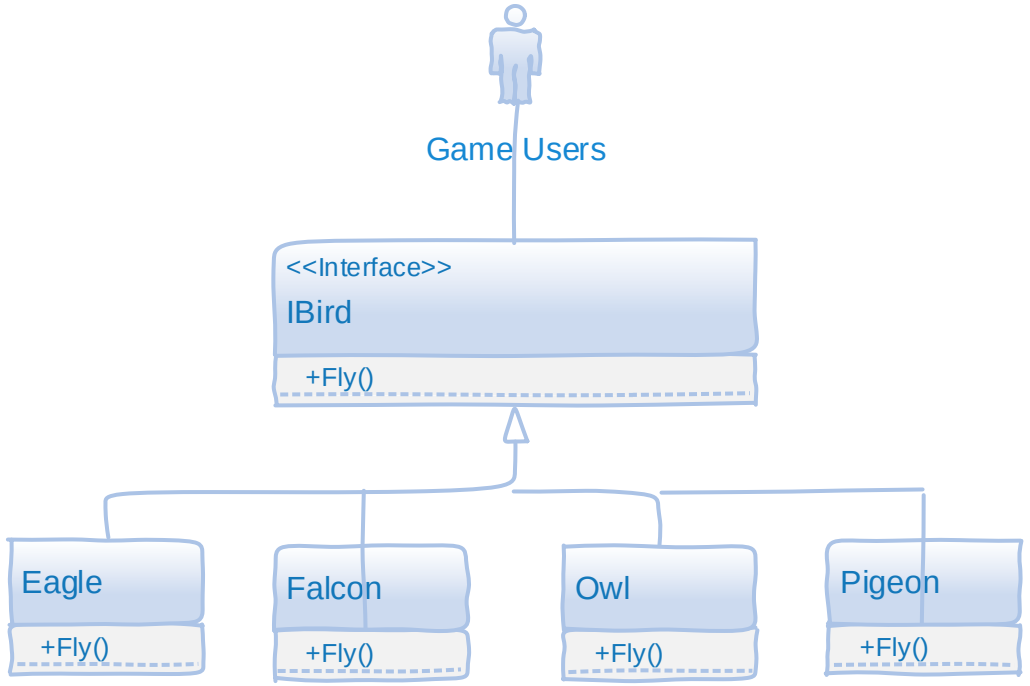
It is hard to read and understand. It does not express its intent well

Liskov Substitution Principle (LSP)

Example

- Suppose you're designing a game that has several bird characters offered via a super interface `IBird` that advertises a *Fly()* method.
- Different bird characters implement this interface and provide *Fly()* implementations based on the way they fly in the game.
- Any usage in the game is via `IBird`. For example:

```
IBird johnEagle = new Eagle();
```



Liskov Substitution Principle (LSP)

Example

- Now, the game is a success and you want to add new bird characters.
- One of this characters is a Penguin!
- What is the problem?

Liskov Substitution Principle (LSP)

Example

Problem

The problem is Penguin cannot fly hence it cannot implement *Fly()* thus it cannot use `IBird` interface!

FLY…???

Liskov Substitution Principle (LSP)

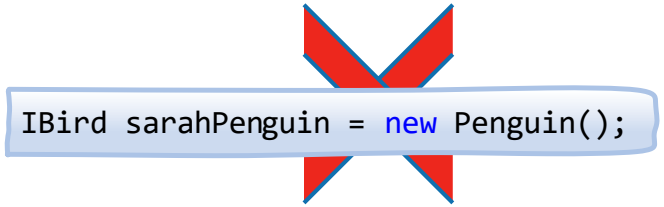
Example

Problem

- LSP indicates that:

"If a super class (or interface) has several subclasses, any object that interacts with the superclass or its interface should be able to instantiate by any of the subclasses without violating any of the super class policies and methods."

- In this example, this is a violation of LSP:



Penguin characters *Walk()* in the game!

Eagle characters *Fly()* in the game!

- Note that we do not want to change object types, i.e. all bird character objects are of type *Ibird.* Otherwise, it's a violation of OCP.

Liskov Substitution Principle (LSP)

Example

Solution

What is the solution for this example that doesn’t violate LSP and OCP?

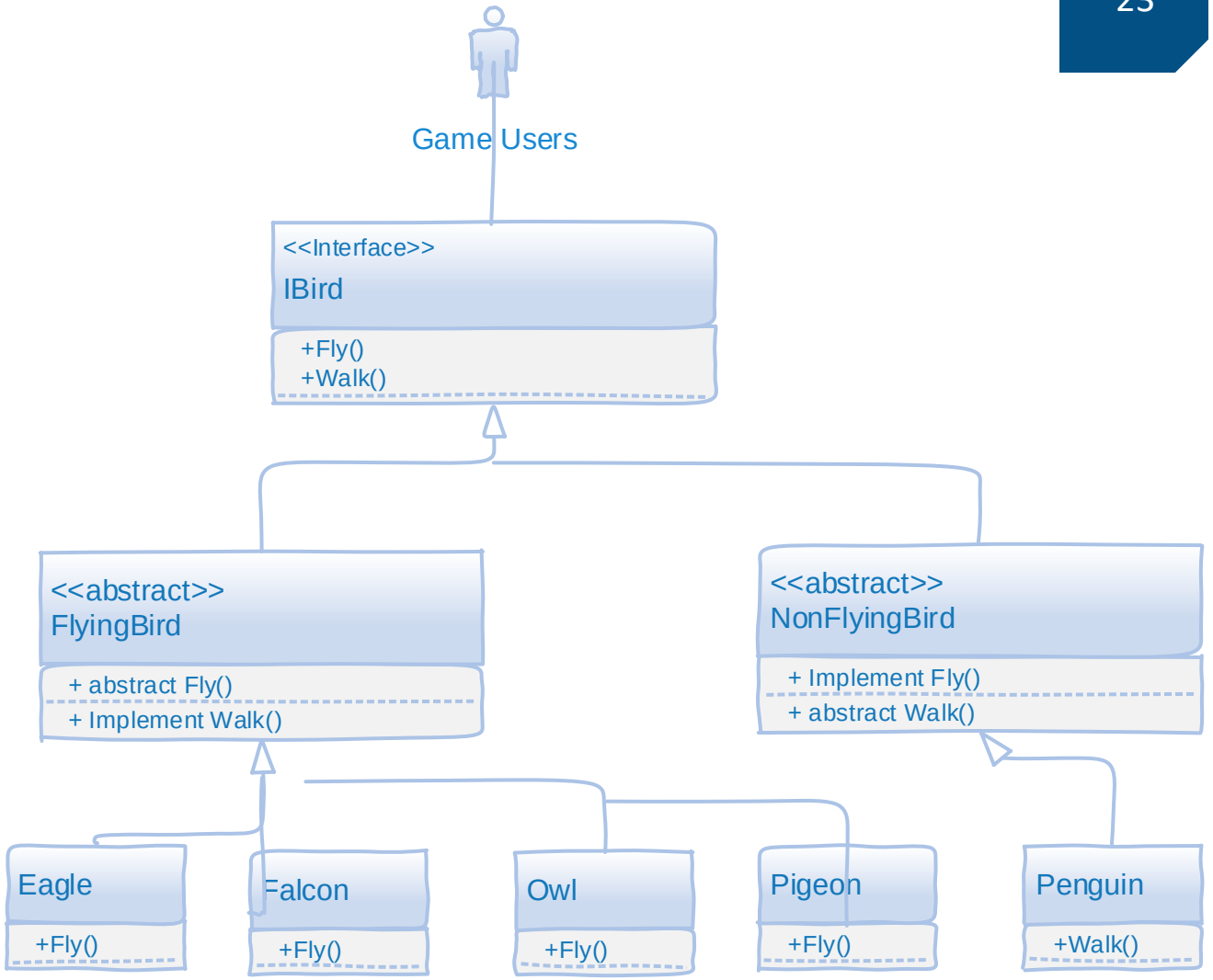
Liskov Substitution Principle (LSP)

Example

Solution

- Two abstract classes are added to implement *IBird*:
1. *FlyingBird* has an implementation for *Walk()* that throws an exception, but only provides an abstraction for *Fly()* leaving its implementation for child classes.
2. *NonFlyingBird* does the opposite.

- All child classes override the abstraction method they implement.
- Note this solution doesn't affect the previous objects of *IBird* so they do not need to change – OCP not violated.
  - As long as *Fly()* signature in *IBird* is not changed, the previous *IBird* usage need no modification.

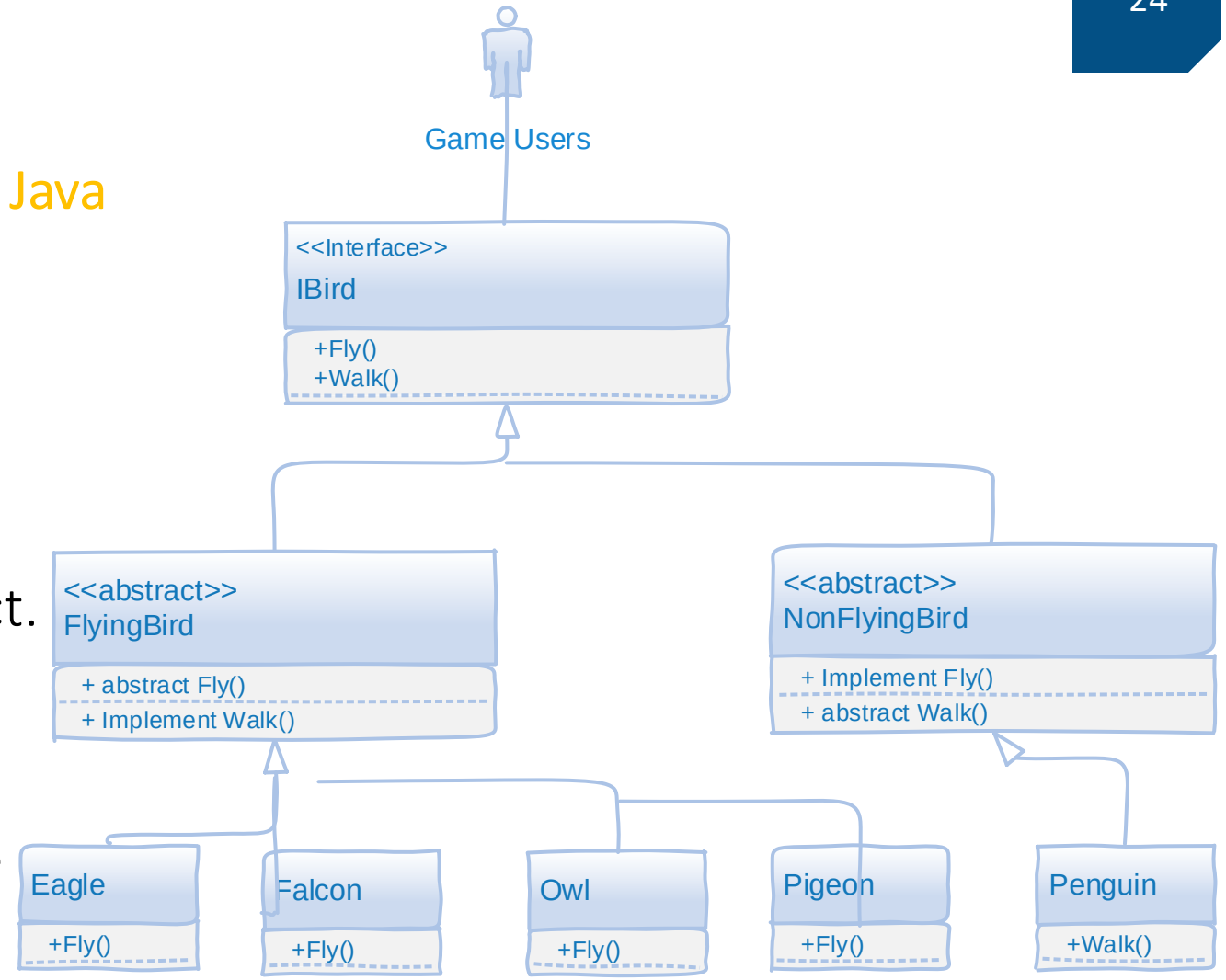


Liskov Substitution Principle (LSP)

Example

Exercise: Implement the given solution in Java

- Note:
  - An abstract class is a class that cannot be initialized by any object.
  - Abstract classes can implement methods.
  - An abstract method doesn't have any implementation. It must be implemented in concrete child classes.



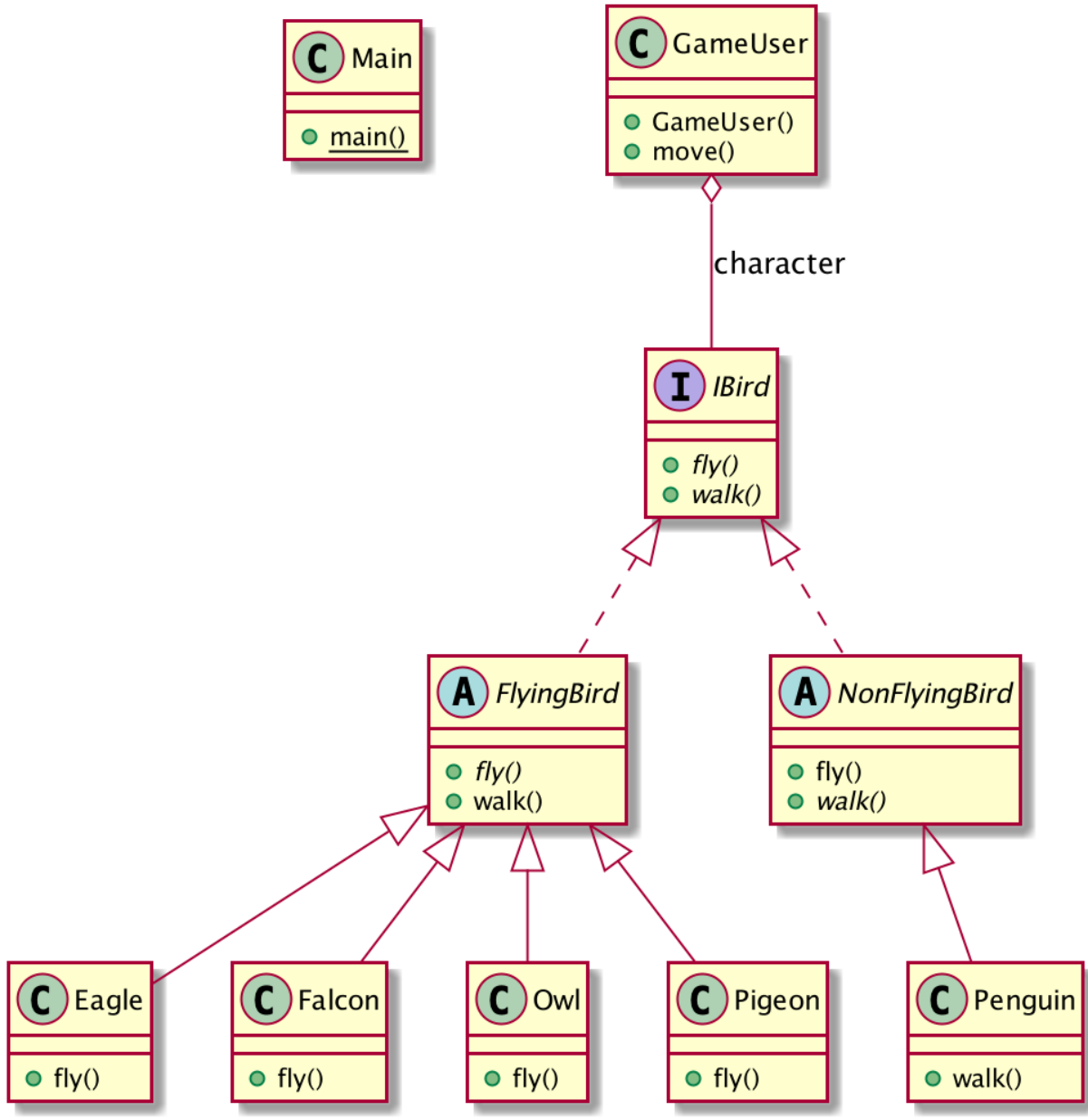
Liskov Substitution Principle (LSP)

Example

Exercise: Implement the given solution in Java

Here's you Main:

```
public static void main(String[] args)
{
    Scanner reader = new Scanner(System.in);

    System.out.println("*************************************");
    System.out.println("Enter 1 for Eagle");
    System.out.println("Enter 2 for Falcon");
    System.out.println("Enter 3 for Owl");
    System.out.println("Enter 4 for Pigeon");
    System.out.println("Enter 5 for Penguin");
    System.out.println("*************************************");

    while (true) {
        // Get player's character
        System.out.print("Please select your character: ");
        int characterCode = reader.nextInt();

        GameUser player = new GameUser(characterCode);
        player.move();
        System.out.println("*************************************");
    }
}
```



Liskov Substitution Principle (LSP)

Example

Exercise: Implement the given solution in Java

A sample program output:

```
**************************************
Enter 1 for Eagle
Enter 2 for Falcon
Enter 3 for Owl
Enter 4 for Pigeon
Enter 5 for Penguin
**************************************
Please select your character: 1
Eagle characters can't walk!
Eagle is flying very high.
**************************************
Please select your character: 3
Owl characters can't walk!
Owl is flying...
**************************************
Please select your character: 5
Penguin walks funny.
Penguin characters don't fly!
**************************************
```

Link to the solution code

Interface Segregation Principle (ISP)

Example

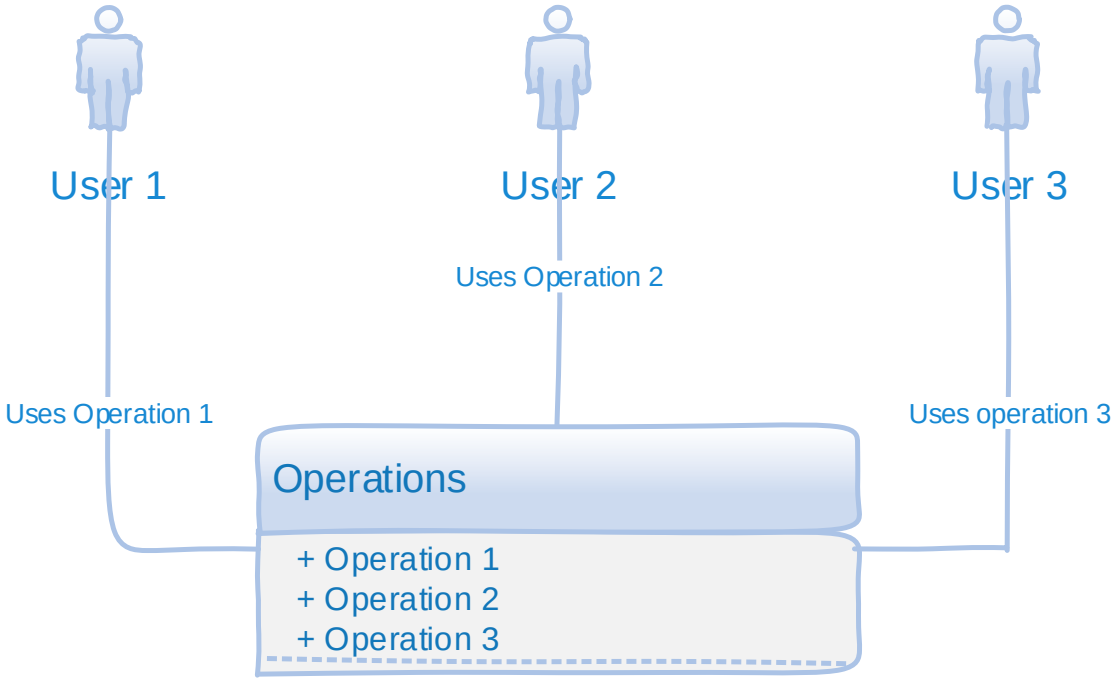
Problem

Consider the following class.

Do you smell rotten design?

Is any of the aforementioned SOLID principles violated?

- Because all actors are coupled to *Operations*, any change to any of the methods reflects on the actors even if they don't use that function.
- This immediately violates SRP - *Operations* has three reasons to change!

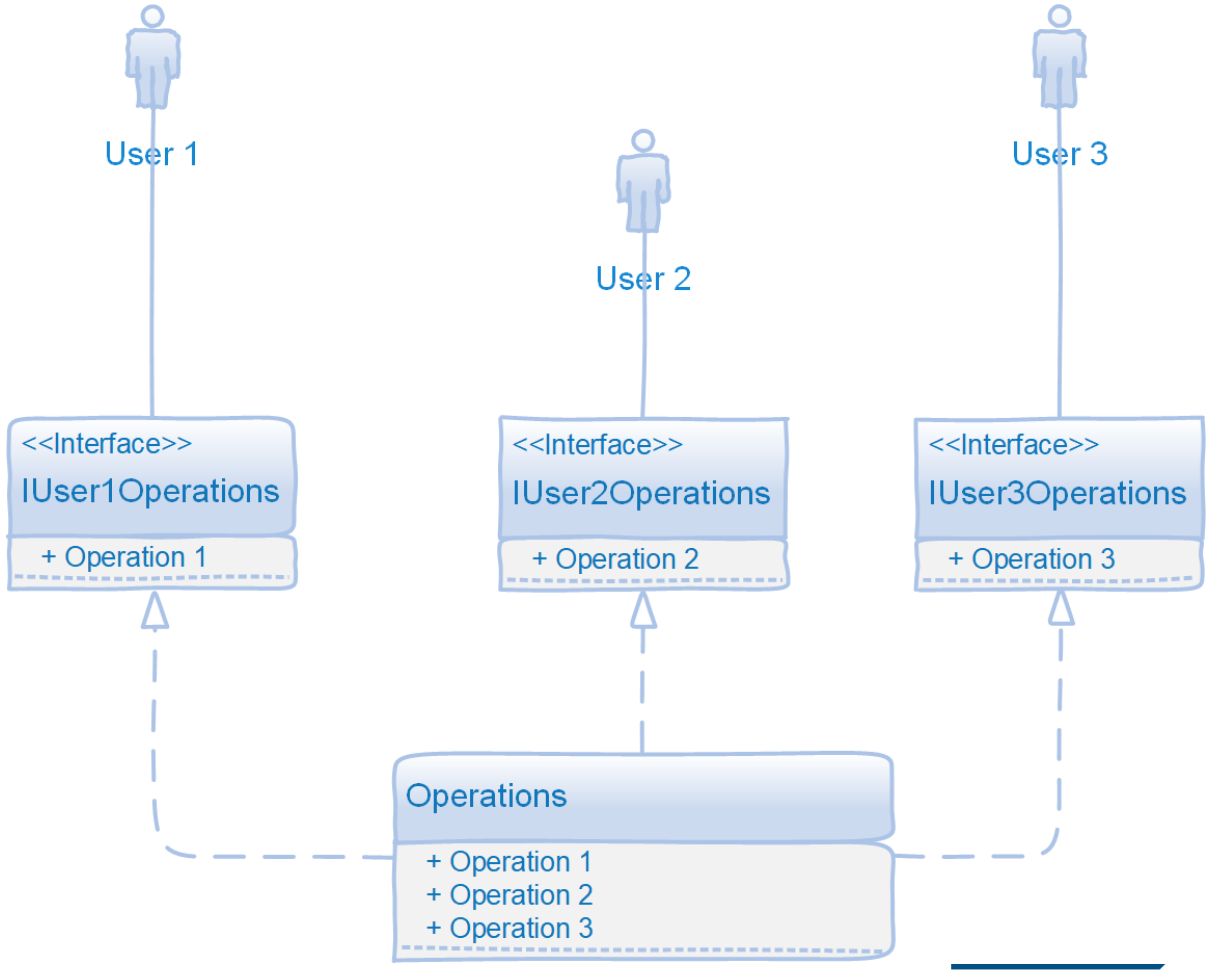


Interface Segregation Principle (ISP)

Example

Solution 1

One solution is to leverage polymorphism and use an interface for each user and put methods that each actor needs in their respective interface.

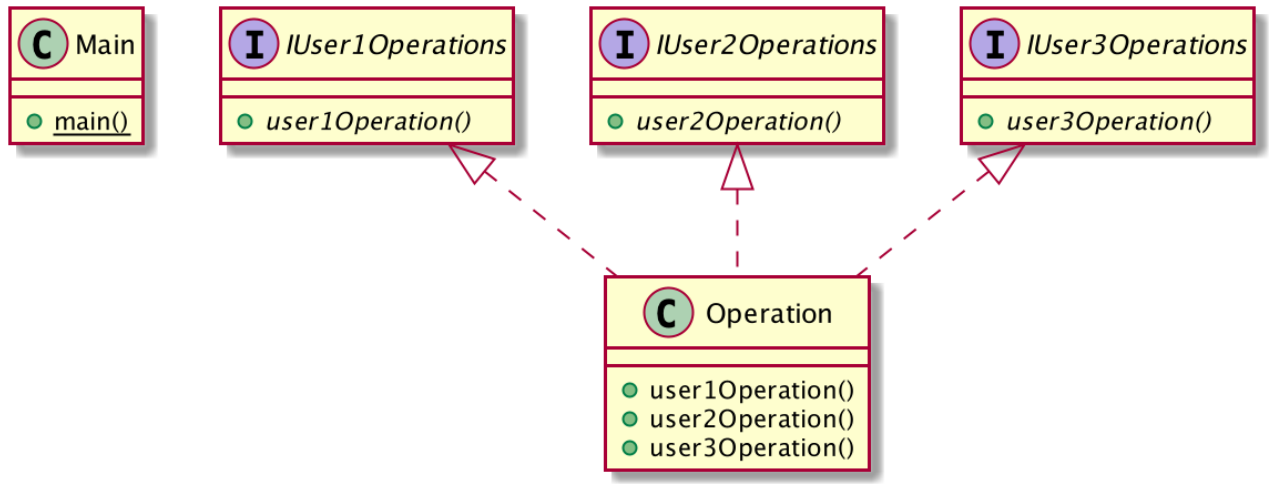


Interface Segregation Principle (ISP)

Example

Solution 1

```
interface IUser1Operations {    void user1Operation();}

interface IUser2Operations {     void user2Operation(); }

interface IUser3Operations {    void user3Operation();}

public class Operation implements IUser1Operations, IUser2Operations, IUser3Operations
{
   public void user1Operation()    {       System.out.println("Performing User 1 operations");    }

   public void user2Operation()    {       System.out.println("Performing User 2 operations");    }

   public void user3Operation()    {       System.out.println("Performing User 2 operations");    }
}

        public static void main(String[] args) {

            IUser1Operations user1;
            IUser2Operations user2;
            IUser3Operations user3;

            user1=new Operation();
            user2 = new Operation();
            user3= new Operation();

            user1.user1Operation();
            user2.user2Operation();
            user3.user3Operation();
        }
```



Interface Segregation Principle (ISP)

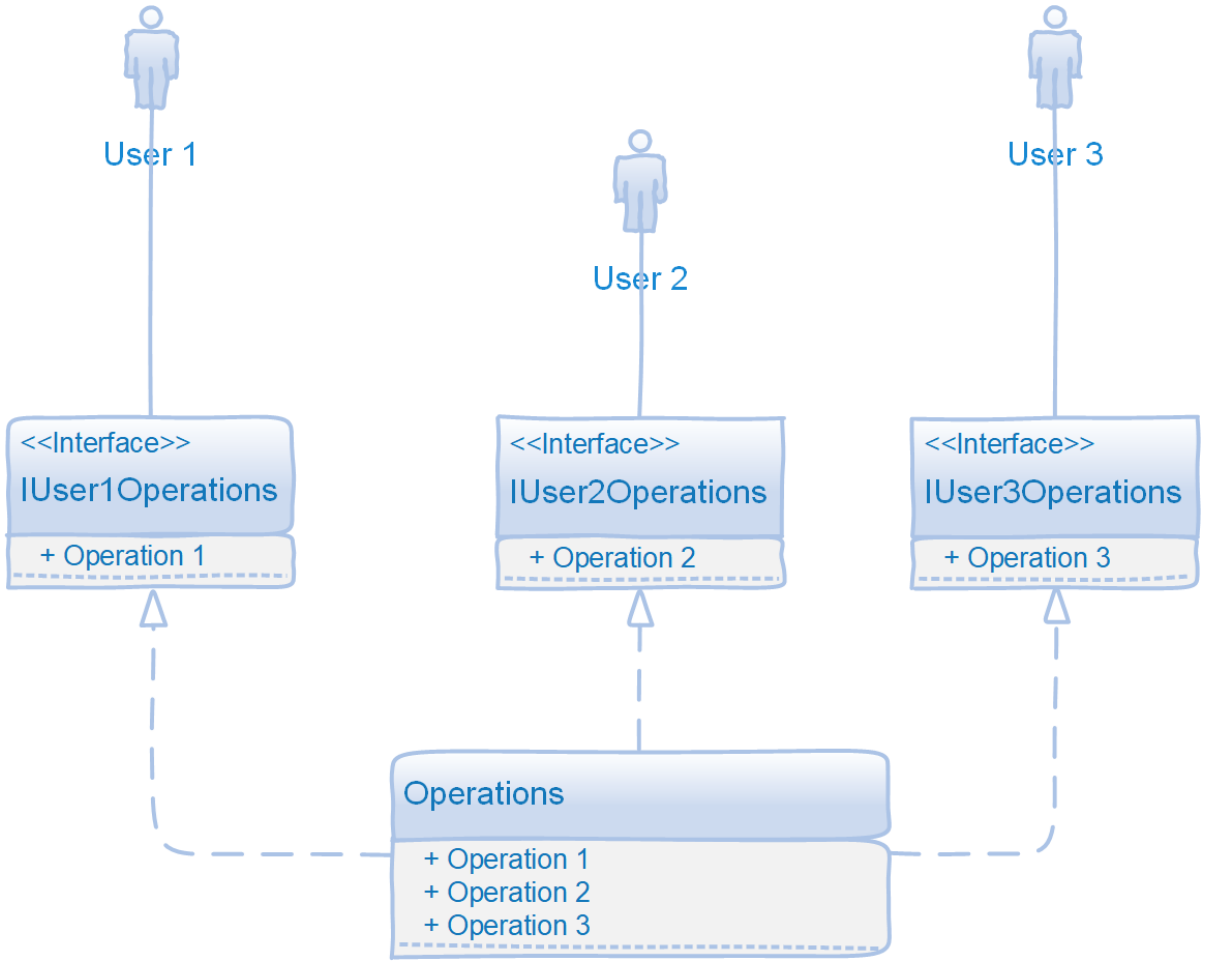
- Applying ISP, instead of having a generic interface for multiple users, we create individual interfaces for each user.
- Moreover, ISP tells us users/actors/classes should not depend on classes that have methods they don't use.

Interface Segregation Principle (ISP)

Example

Solution 1 – Problem?

- Using this solution, we're limiting each user to only be able to see and use its own operation.
- Have we solved the smell(s)?
  - We can argue that this solution still violates SRP as *Operations* have three reasons to change.
  - Currently each Operations method is independent of the others but it's very likely that as the software evolves, coupling between methods start to emerge.
  - Moreover, the current solution still violates the second ISP clause as users still depend on a class with methods that they don't use.



Interface Segregation Principle (ISP)

Example

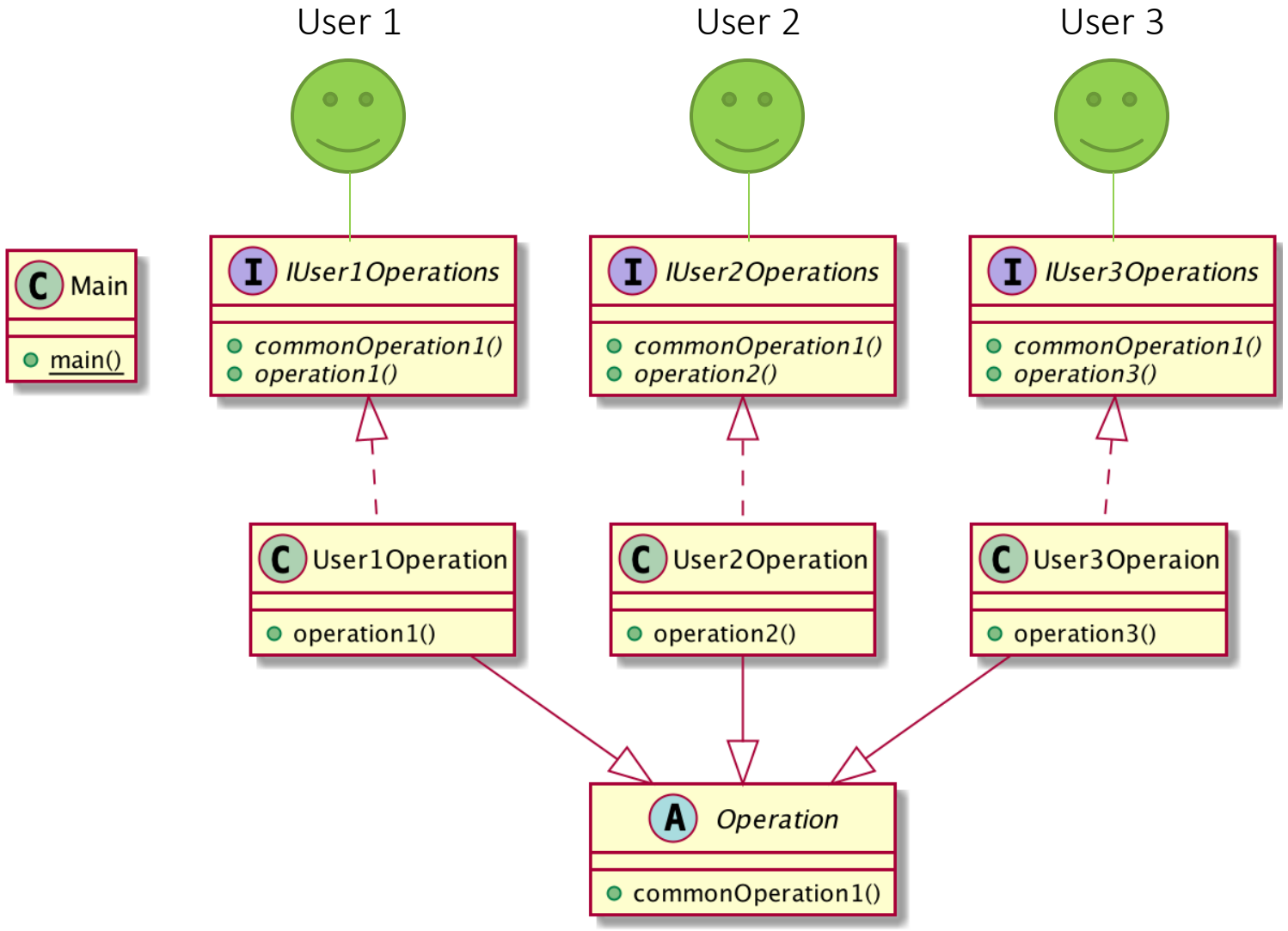
Solution 2

What would be a solution that doesn't violate both SRP and ISP? Design this second solution.

Interface Segregation Principle (ISP)

Example

Solution 2



Interface Segregation Principle (ISP)

On the component level, the Common Reuse Principle (CRP) relates to ISP and advises on "don’t force users of a component to depend on things (classes) they don’t use". Or simply, "don’t depend on things you don’t need".

Dependency Inversion Principle (DIP)

"No class should be associated with concrete classes. Always associate classes with interfaces for other classes."

DIP accommodates change since interfaces are less likely to change but concreate classes are likely to change.

Dependency Injection

- SOLID principles promote loosely coupled components.
- One of the benefits of SOLID is Dependency Injection can be easily implemented, that is we associate objects with interfaces that can be initialized by any of its child classes.
- The intent behind dependency injection is to decouple objects to the extent that no client code has to be changed simply because an object it depends on needs to be changed to a different one*.

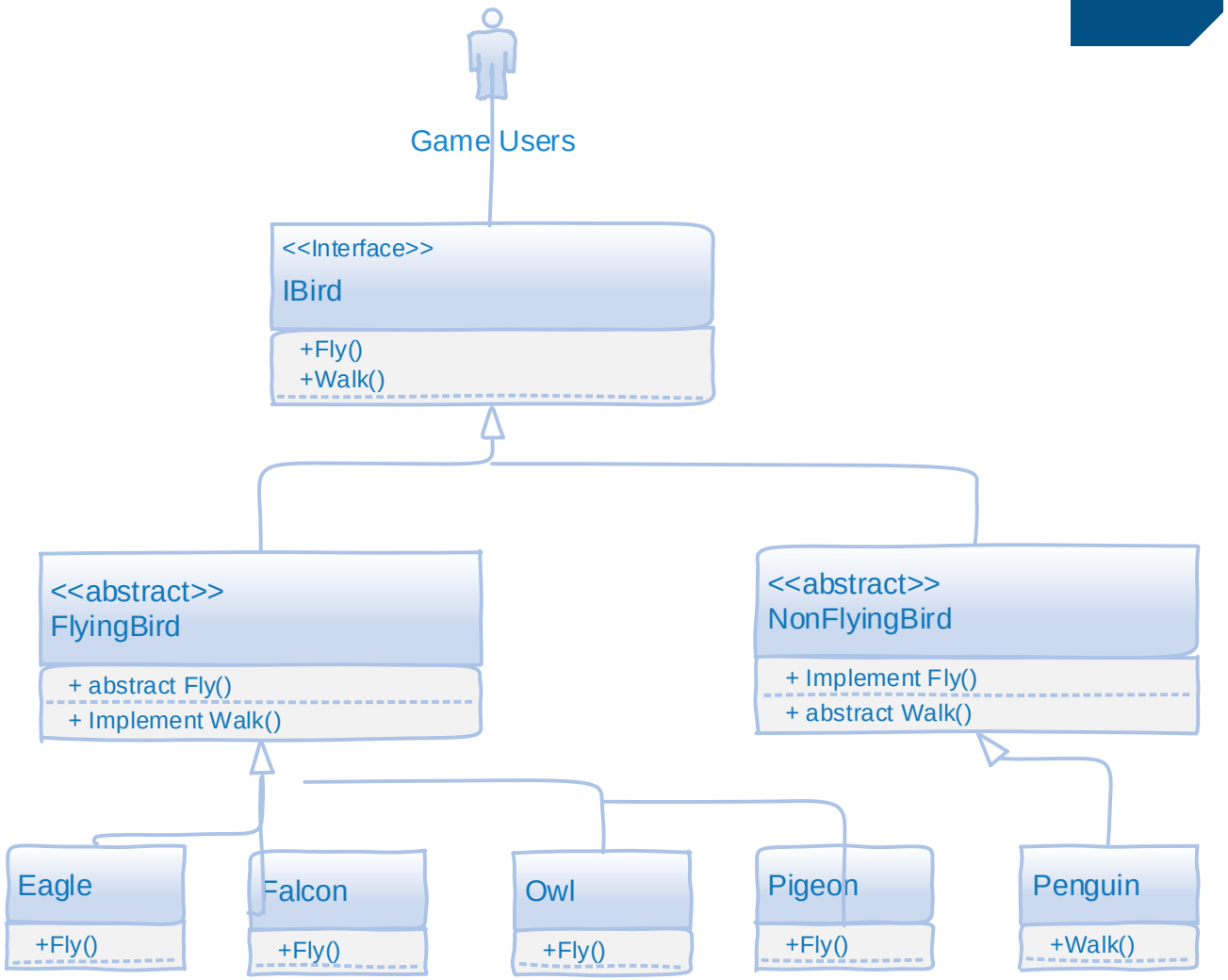
From https://en.wikipedia.org/wiki/Dependency_injection

Dependency Injection

Example

More Information

Recalling from this example, any object user of *IBird* can be initialized by any of the subclasses without requiring to change the code.



Dependency Injection

Example

More Information

Object type identification is postponed until its initialization.

```
public class GameUser
{
    IBird character;

    public GameUser(int characterCode)
    {
        switch (characterCode) {
            case 1:
                character = new Eagle();
                break;
            case 2:
                character = new Falcon();
                break;
            case 3:
                character = new Owl();
                break;
            case 4:
                character = new Pigeon();
                break;
            case 5:
                character = new Penguin();
                break;
        }
    }
...
```

Object is declared here, but its actual type will be identified when it's initialized, allowing significant flexibility.