SOFTWARE DESIGN TECHNIQUES (CSCN72040)

Week-4

SOLID Design Principles

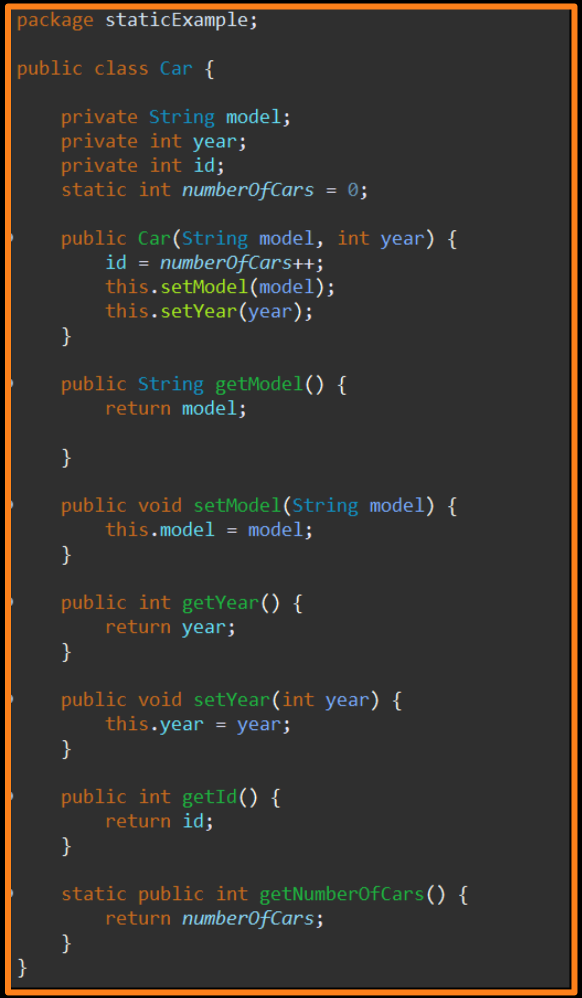
Petros Spachos

CONESTOGA

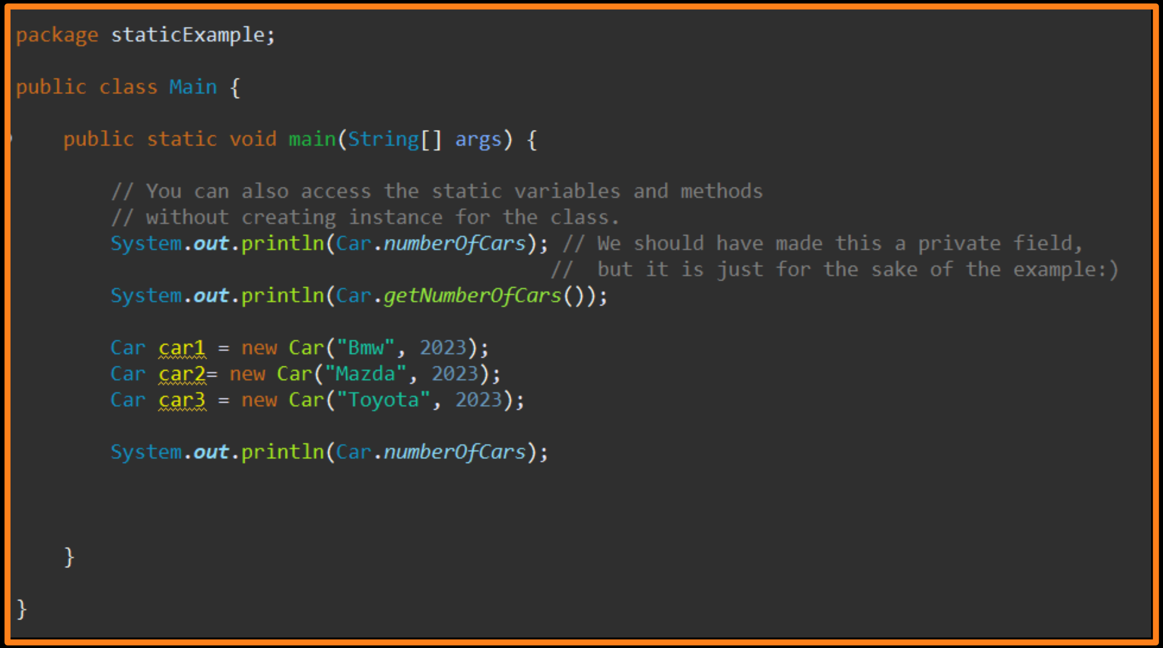
Review

STATIC KEYWORD

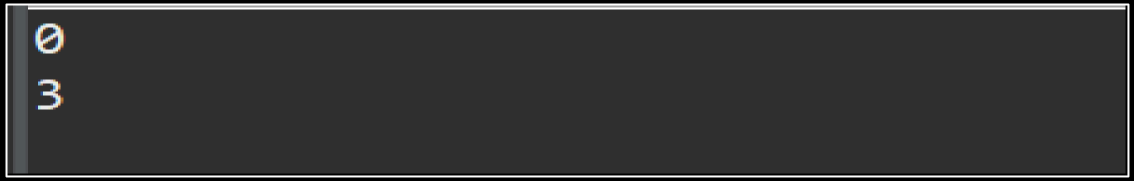
The static keyword is used to create fields and methods that belong to the class, rather than to an instance of the class [1].



```
package staticExample;

public class Car {

    private String model;
    private int year;
    private int id;
    static int numberOfCars = 0;

    public Car(String model, int year) {
        id = numberOfCars++;
        this.setModel(model);
        this.setYear(year);
    }

    public String getModel() {
        return model;

    }

    public void setModel(String model) {
        this.model = model;
    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public int getId() {
        return id;
    }

    static public int getNumberOfCars() {
        return numberOfCars;
    }
}
```



```
package staticExample;

public class Main {

    public static void main(String[] args) {

        // You can also access the static variables and methods
        // without creating instance for the class.
        System.out.println(Car.numberOfCars); // We should have made this a private field,
                                              //  but it is just for the sake of the example:)
        System.out.println(Car.getNumberOfCars());

        Car car1 = new Car("Bmw", 2023);
        Car car2= new Car("Mazda", 2023);
        Car car3 = new Car("Toyota", 2023);

        System.out.println(Car.numberOfCars);



    }

}
```



```
0
3
```

Final KEYWORD

The final keyword can be used with variables , method and classes

final primitive type **variable**

Once you initialize it , you cannot modify it

final reference type **variable**

Once you initialize it , you cannot refer it to another object

final **method**

A final method cannot be overridden

final **class**

A final class cannot be subclassed

ABSTRACT CLASS VS INTERFACE

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces [1].

Which should you use, abstract classes or interfaces?

Consider using abstract classes if any of these statements apply to your situation[1]:

- You want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
- You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Consider using interfaces if any of these statements apply to your situation [1]:

- You expect that unrelated classes would implement your interface. For example, the interfaces Comparable is implemented by many unrelated classes.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
- You want to take advantage of multiple inheritance of type.

INTERFACE

Default Methods

Default methods enable you to add new functionality to the interfaces of your libraries and ensure compatibility with code written for older versions of those interfaces [1] .

WRAPPER CLASSES

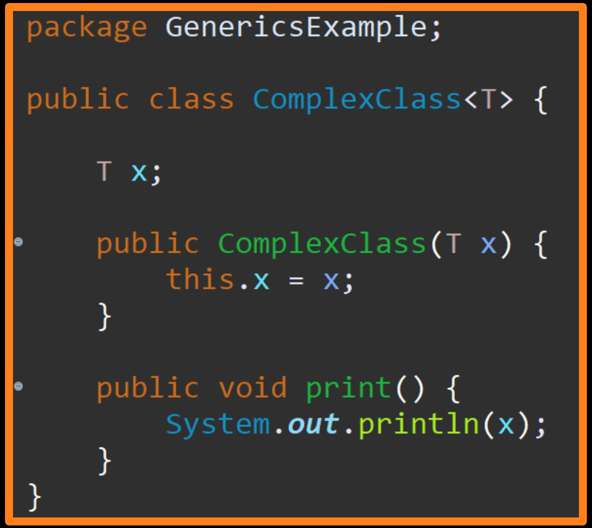
Wrapper classes provide a way to use primitive data types as objects. The following table lists the primitive types and their corresponding wrapper classes:

- Data structures in the Collection framework such as ArrayList works only with objects
- Generic do not allow using primitive types

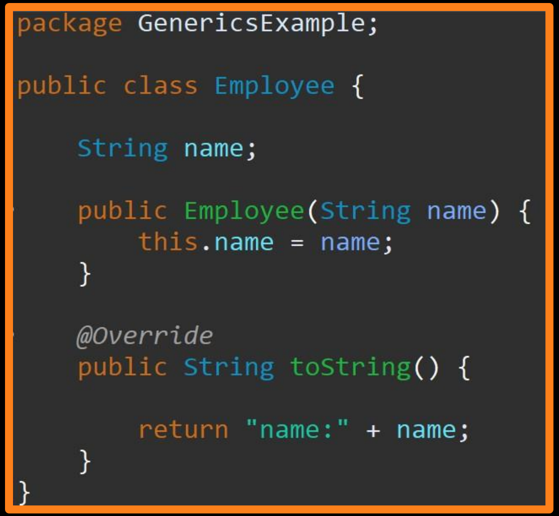
Primitive type	**Wrapper class**
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

JAVA GENERIC

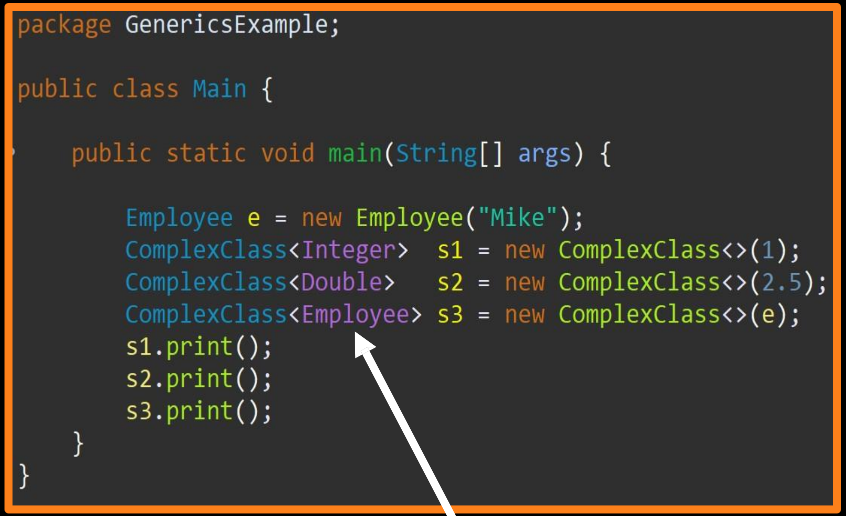
Mainly generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Type parameters provide a way for you to re-use the same code with different inputs.



```java
package GenericsExample;

public class ComplexClass<T> {

    T x;

    public ComplexClass(T x) {
        this.x = x;
    }

    public void print() {
        System.out.println(x);
    }
}
```



```java
package GenericsExample;

public class Employee {

    String name;

    public Employee(String name) {
        this.name = name;
    }

    @Override
    public String toString() {

        return "name:" + name;
    }
}
```



```java
package GenericsExample;

public class Main {

    public static void main(String[] args) {

        Employee e = new Employee("Mike");
        ComplexClass<Integer>  s1 = new ComplexClass<>(1);
        ComplexClass<Double>   s2 = new ComplexClass<>(2.5);
        ComplexClass<Employee> s3 = new ComplexClass<>(e);
        s1.print();
        s2.print();
        s3.print();
    }
}
```

Different types as parameters

ARRAYS, LIST AND ARRAYLIST

Arrays

It is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed [1].

The List Interface

A List is an ordered Collection. Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for the following [1]:

Positional access — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.
Search — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.
Iteration — extends Iterator semantics to take advantage of the list's sequential nature. The listIterator methods provide this behavior.
Range-view — The sublist method performs arbitrary range operations on the list.

ArrayList

Resizable-array implementation of the List interface[1].

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html

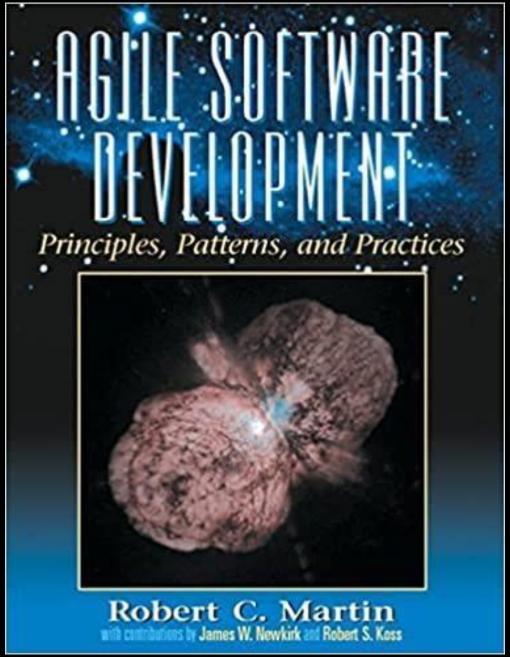
SOLID Design Principles

OUTLINE

- Introduction
- SOLID Design Principles
- Single Responsibility Principle (SRP)
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle
- Demo
- Summary and Conclusion

INTRODUCTION

SOLID PRINCIPLES


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

Robert C. Martin, called Uncle Bob

He is a co-author of the Agile Manifesto.

S.O.L.I.D Principles

- The SOLID principles are a set of golden rules that aim to improve the design and maintainability of software.

- These principles were first introduced in the early 2000s and have since become widely accepted as best practices for developers working with object-oriented programming languages.

- SOLID principles are particularly relevant for agile development, as they help create flexible, scalable, and easy to modify code.

Advantages to follow SOLID principles

- Improved maintainability: You can create code that is easier to maintain and modify over time because the SOLID principles encourage the creation of modular, flexible code that is less prone to errors and more resistant to changes in requirements.
- Reduced complexity: The SOLID principles help to reduce the complexity of software by promoting the use of abstraction and encapsulation, which can make it easier to understand and work with the code.
- Enhanced flexibility: These principles encourage the creation of flexible code that is open to extension but closed to modification, which encourages flexibility without breaking existing functionality.
- Increased scalability: The SOLID principles can help to make software more scalable, as they encourage the use of abstractions and decoupled dependencies, which can help to prevent the codebase from becoming overly complex and difficult to manage.

What are SOLID principles?

- SOLID is a mnemonic device for 5 design principles of object-oriented programs (OOP) that result in readable, adaptable, and scalable code. SOLID can be applied to any OOP program.

- The 5 principles of SOLID are:

1. **S**ingle-responsibility principle
2. **O**pen-closed principle
3. **L**iskov substitution principle
4. Interface segregation principle
5. **D**ependency inversion principle

Single Responsibility Principle

SOLID PRINCIPLES

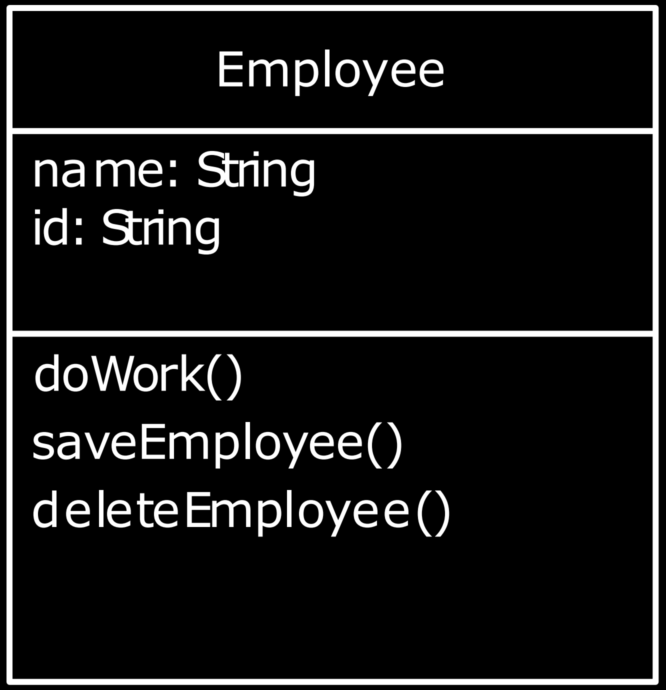
Single Responsibility Principle

"A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class."
Robert C. Martin

A class and a method should have one, and only one, reason to change.

SOLID PRINCIPLES

Single Responsibility Principle (SRP)



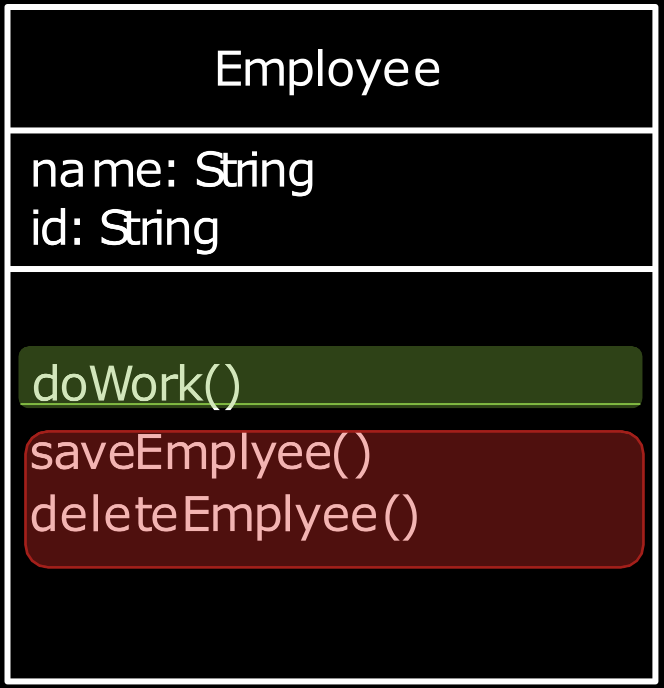
Employee

name: String
id: String

doWork()
saveEmployee()
deleteEmployee()

SOLID PRINCIPLES

Single Responsibility Principle (SRP)



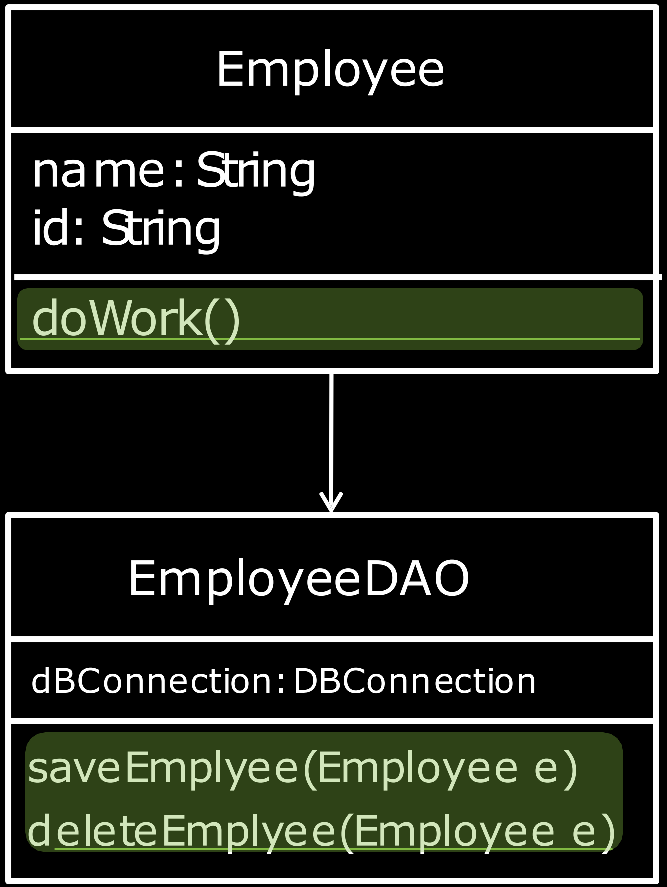
Employee

name: String
id: String

doWork()

saveEmplyee()
deleteEmplyee()

🙁



Employee

name: String
id: String

doWork()

EmployeeDAO

dBConnection: DBConnection

saveEmplyee(Employee e)
deleteEmplyee(Employee e)

Responsibility:
Handle core employee related tasks

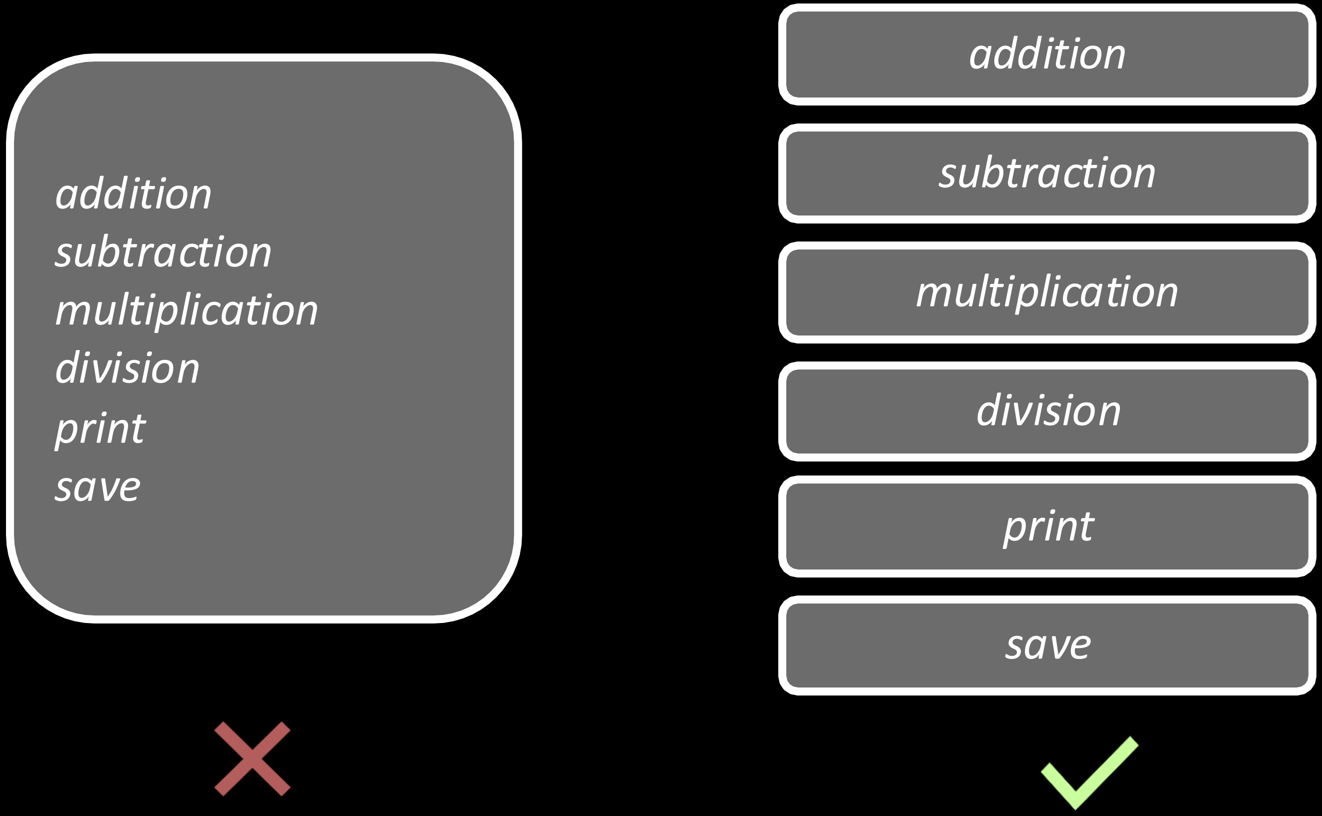
🙂

Responsibility:
Handle Database operations

SOLID PRINCIPLES

Single Responsibility Principle (SRP)

Function



addition
subtraction
multiplication
division
print
save

✗

addition

subtraction

multiplication

division

print

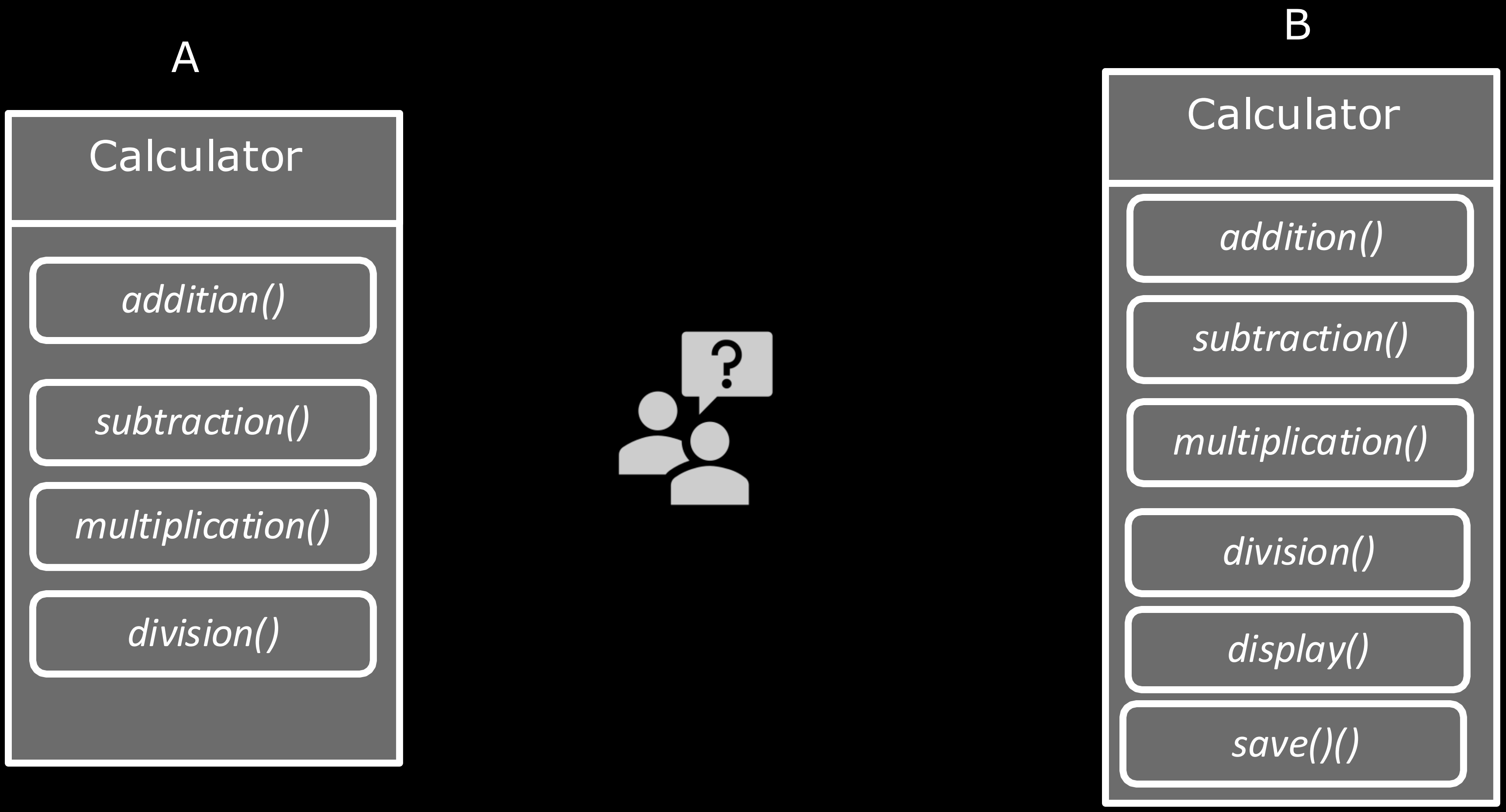
save

✓

SOLID PRINCIPLES

Single Responsibility Principle (SRP)

Class



A

Calculator

addition()

subtraction()

multiplication()

division()

B

Calculator

addition()

subtraction()

multiplication()

division()

display()

save()()

Open/Closed Principle

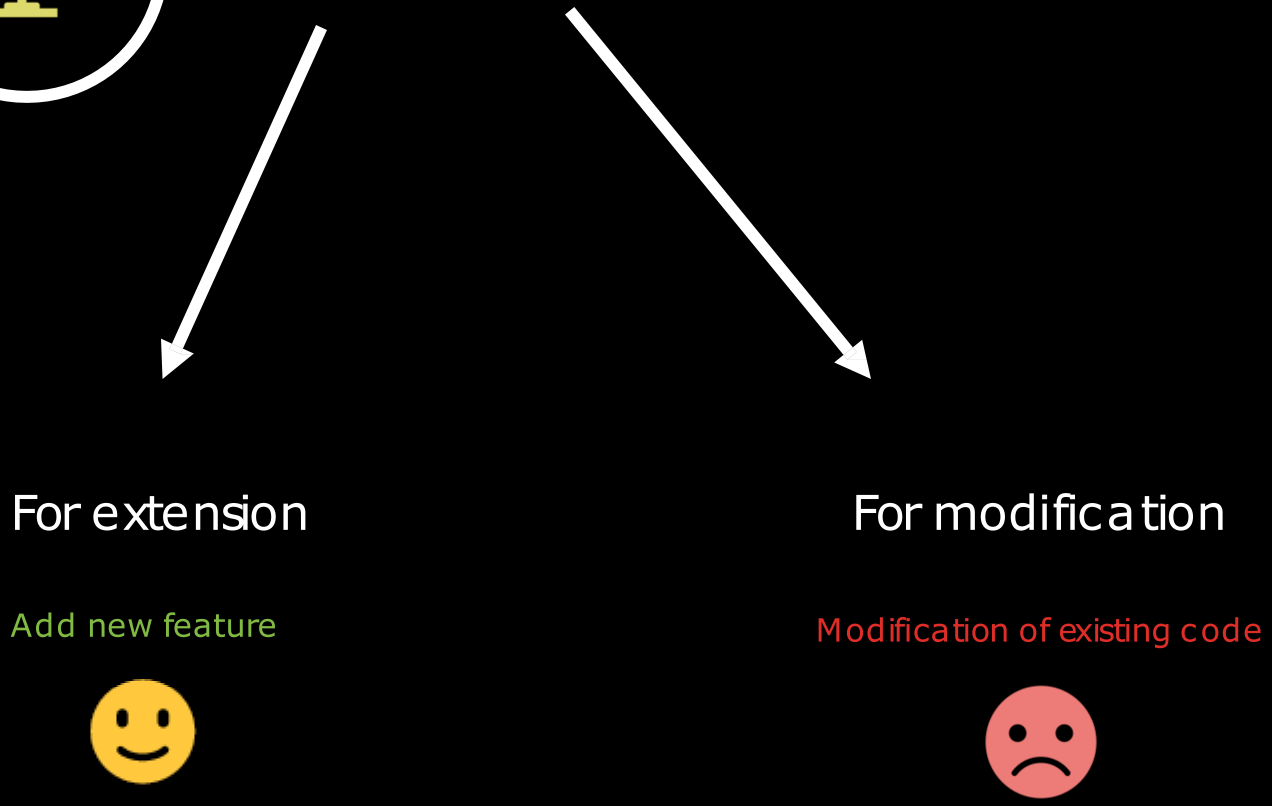
SOLID PRINCIPLES

Open/Closed Principle

“Software entities … should be open for extension, but closed for modification.”
Robert C. Martin

SOLID PRINCIPLES

Open/Closed Principle



For extension

Add new feature

🙂

For modification

Modification of existing code

☹

SOLID PRINCIPLES

Open/Closed Principle

```
calculateArea(… . shapes [])
{
….
 totalArea = 0
for shape in shapes {
…….
If isSquare(shape) {
totalArea += shape.side * shape.side
}




return totalArea
}
```

SOLID PRINCIPLES

Open/Closed Principle

Modification of existing code

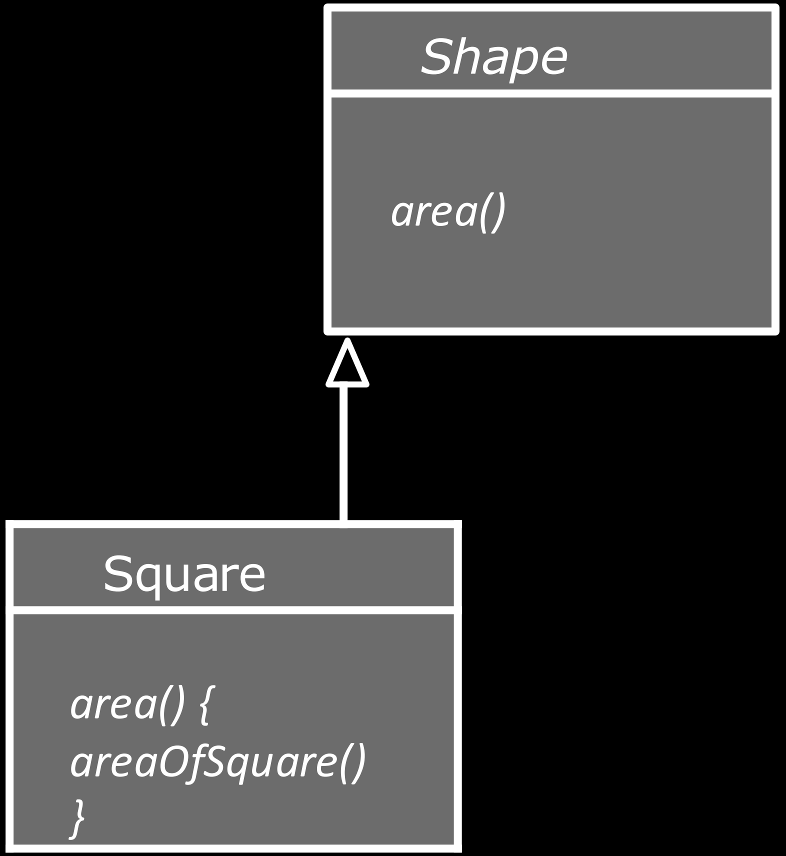
☹

```
calculateArea(…. shapes [])
{
….
 totalArea = 0
for shape in shapes {
…….
If isSquare(shape) {
totalArea += shape.side *shape.side
}
else If isSCircle(shape) {
totalArea += shape.radius * shape.radius * Pi
}
… …
return totalArea
}
```

SOLID PRINCIPLES

Open/Closed Principle

```
calculateArea(… . shapes [])
{
….
 totalArea = 0
for shape in shapes {
…
totalArea += shape.area()
…
}
return totalArea
}
```

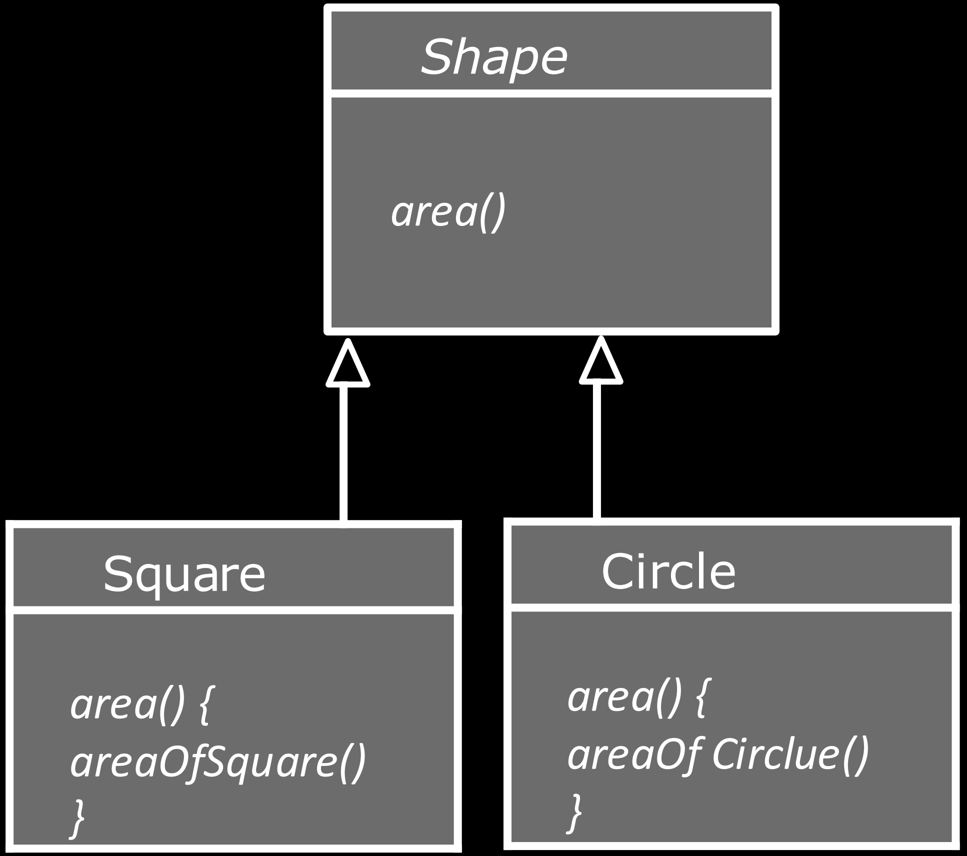


SOLID PRINCIPLES

Open/Closed Principle

No change

```
calculateArea(… . shapes [])
{
….
 totalArea = 0
for shape in shapes {
…
totalArea += shape.area()
…
}
return totalArea
}
```



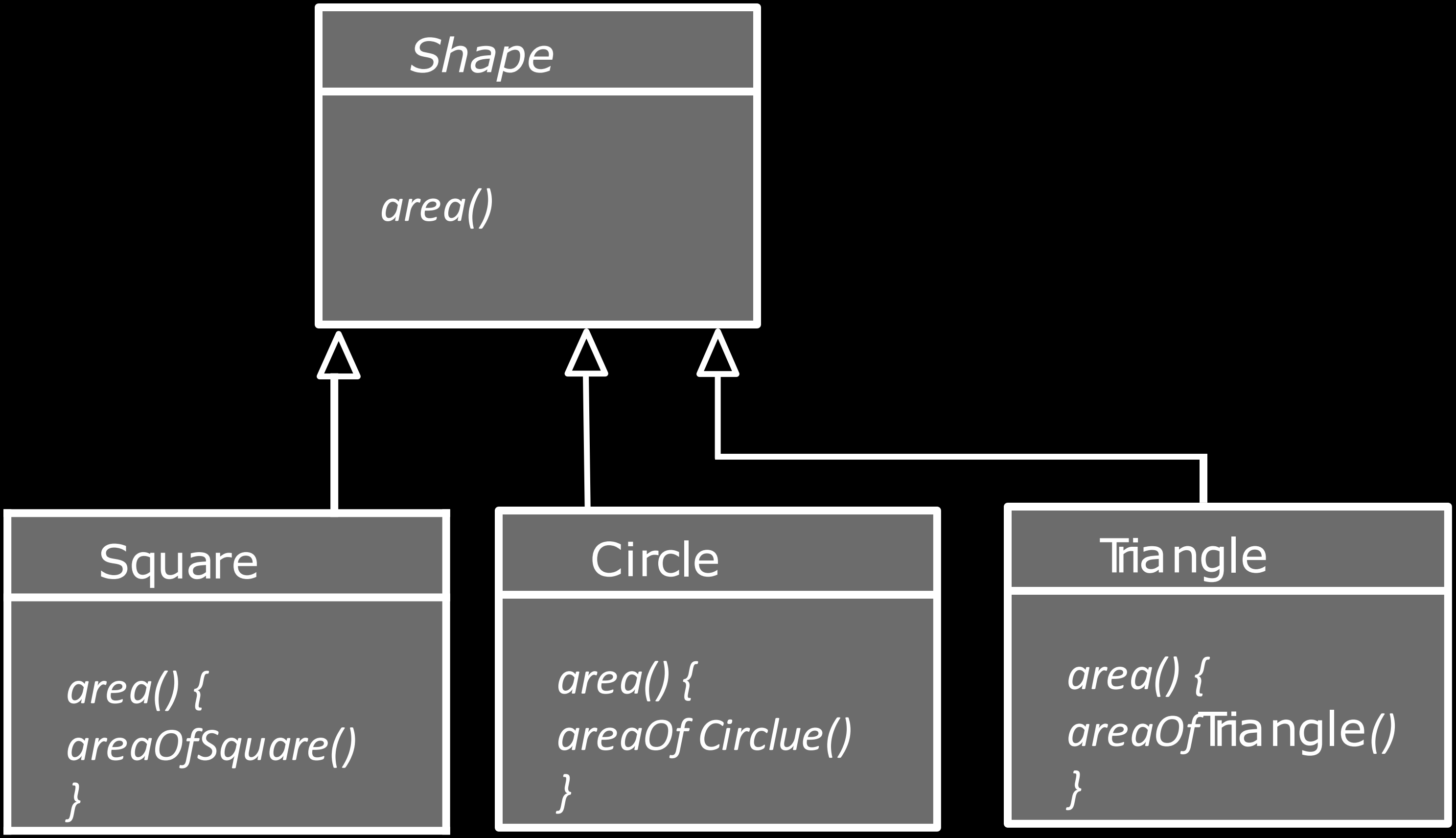
SOLID PRINCIPLES

Open/Closed Principle

No change

🙂

```
calculateArea(... . shapes [])
{
....
 totalArea = 0
for shape in shapes {
...
totalArea += shape.area()
...
}
return totalArea
}
```



Shape

area()

Square

area() {
areaOfSquare()
}

Circle

area() {
areaOf Circlue()
}

Triangle

area() {
areaOfTriangle()
}

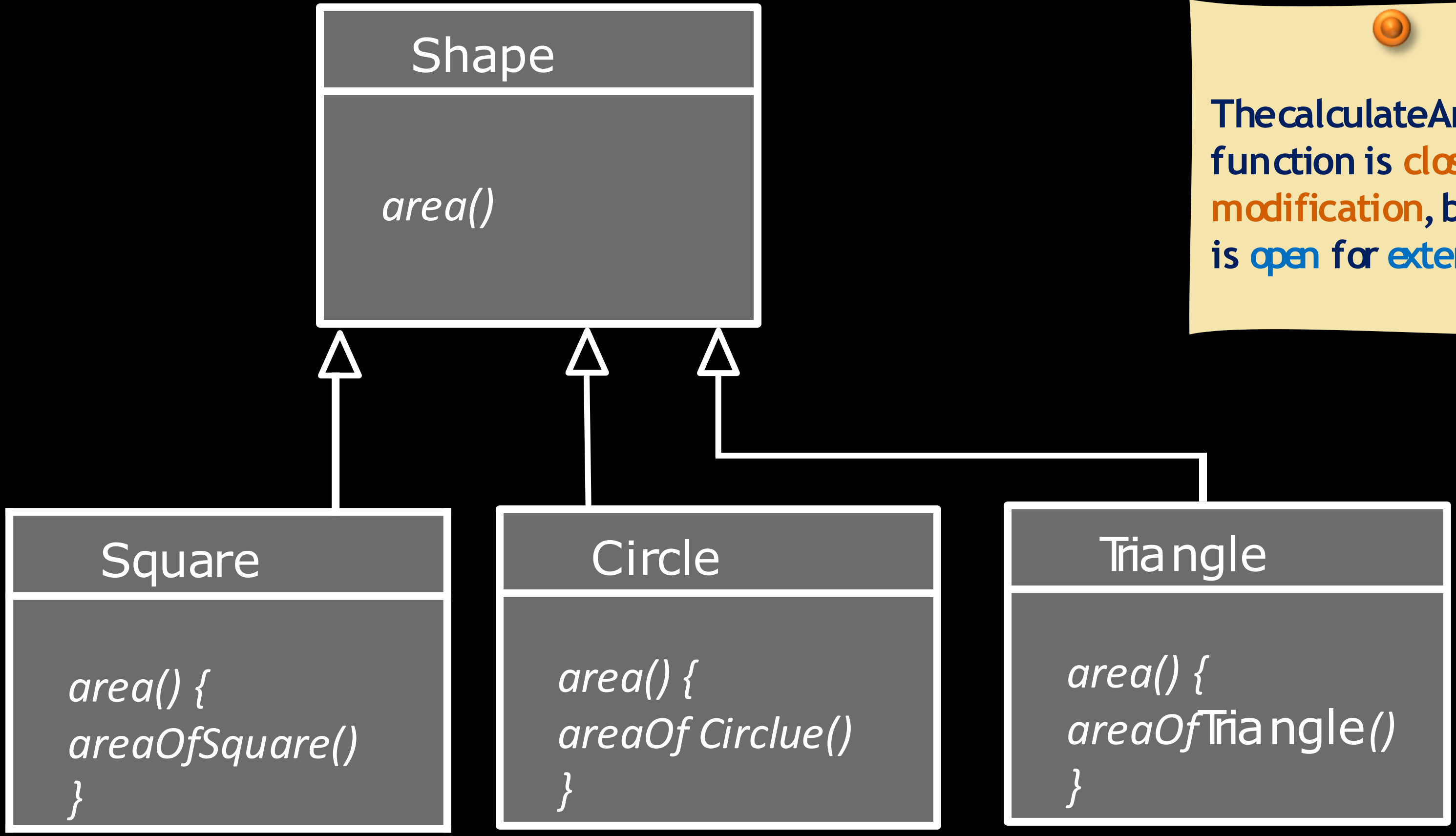
SOLID PRINCIPLES

Open/Closed Principle

No change

🙂

```
calculateArea(….shapes [])
{
….
 totalArea =0
for shape in shapes {
…
totalArea +=shape.area()
…
}
return totalArea
}
```



Shape

area()

Square

area() {
areaOfSquare()
}

Circle

area() {
areaOf Circlue()
}

Triangle

area() {
areaOfTriangle()
}

The calculateArea(..) function is close for modification, but it is open for extension

Liskov Substitution Principle

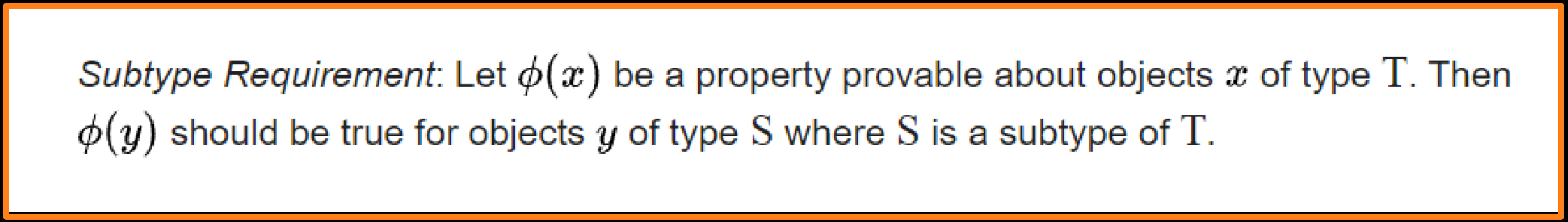
Let’s start with this video

https://www.youtube.com/watch?v=-Z-17h3jG0A

SOLID PRINCIPLES

Liskov Substitution Principle

Liskov and Jeannette Wing described the principle succinctly in a 1994 paper as follows [1]



Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

SOLID PRINCIPLES

Liskov Substitution Principle

Subtypes must be substitutable for their base types

if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program

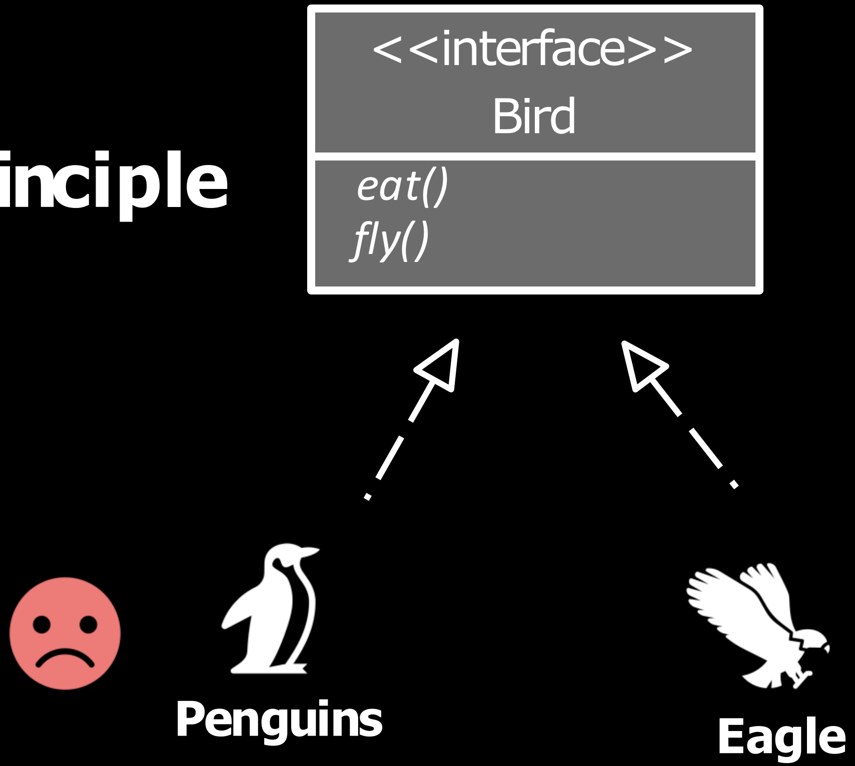
SOLID PRINCIPLES

Liskov Substitution Principle

"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."
Robert c. Martin

SOLID PRINCIPLES

Liskov Substitution Principle



<<interface>>

Bird

eat()

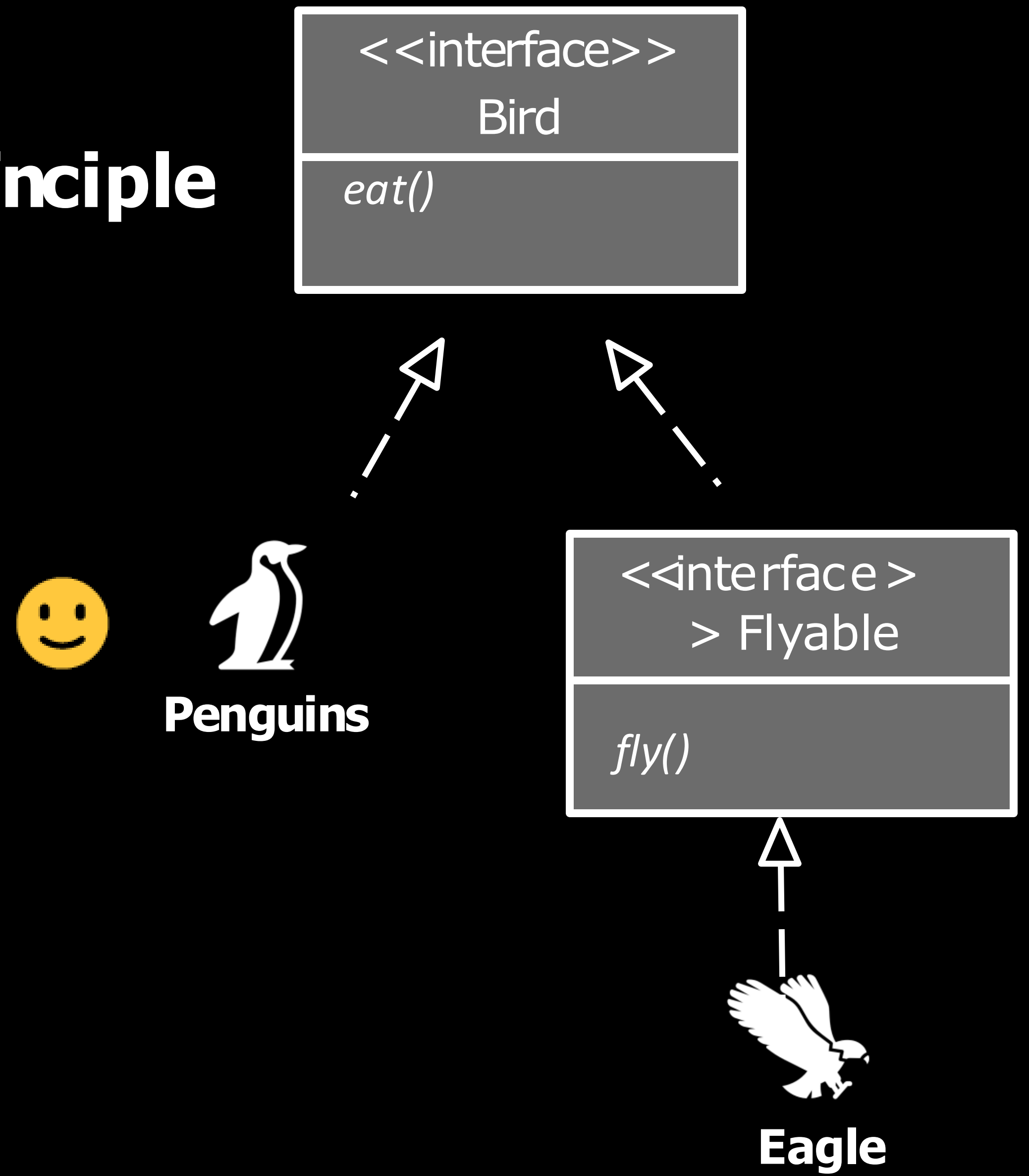
fly()

Penguins

Eagle

SOLID PRINCIPLES

Liskov Substitution Principle



<<interface>>
Bird

eat()

Penguins 🙂

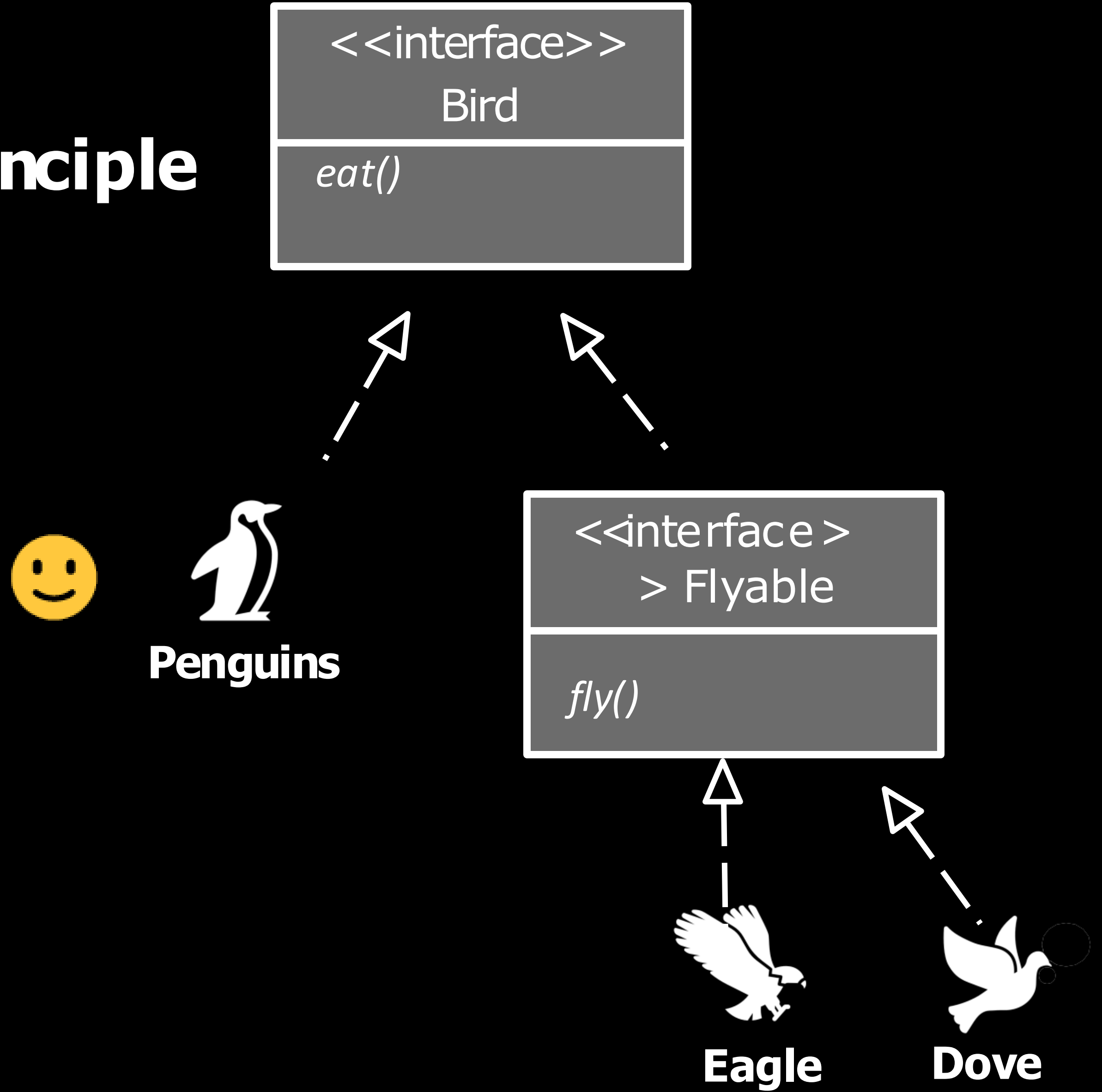
<<interface>
> Flyable

fly()

Eagle

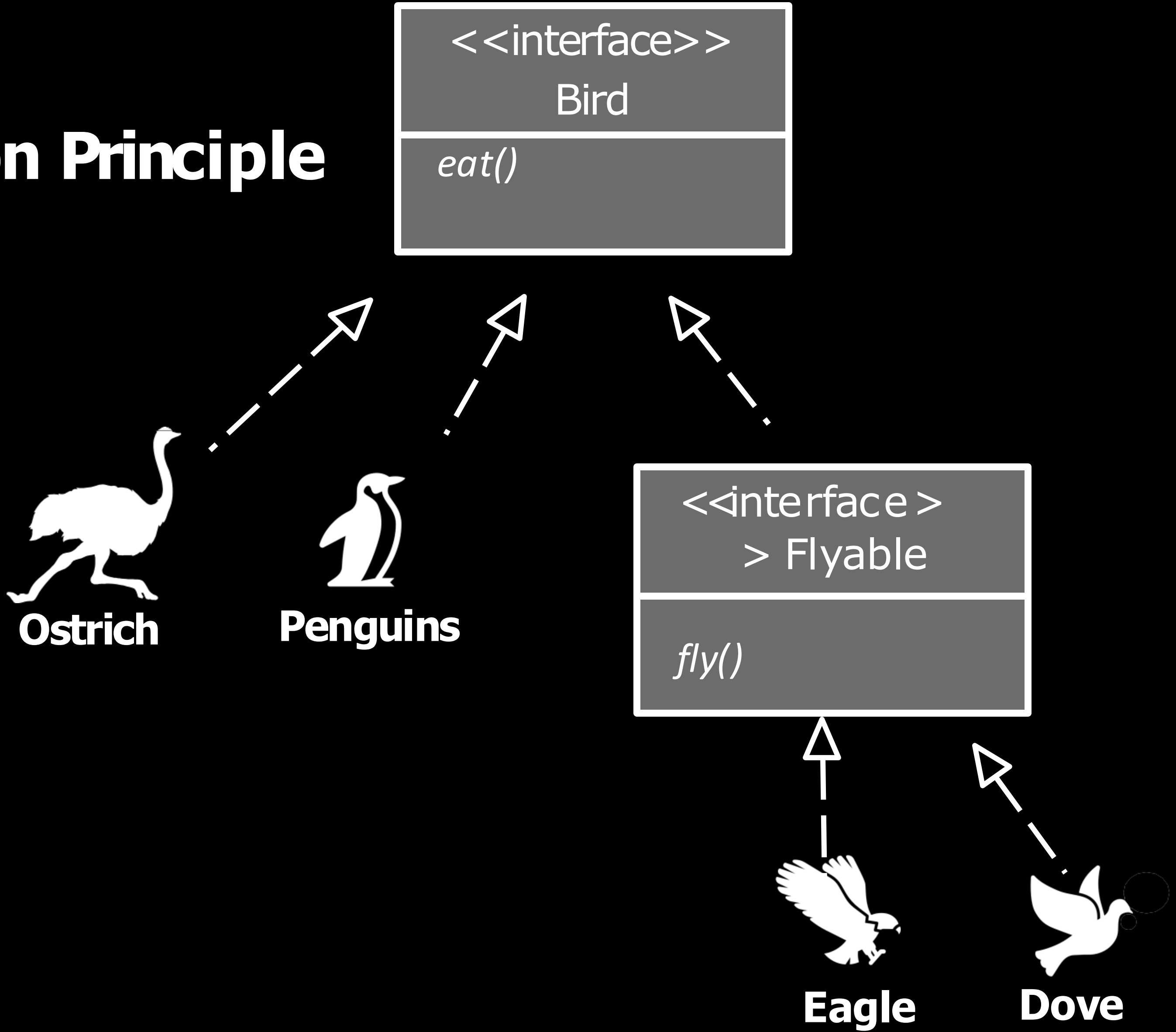
SOLID PRINCIPLES

Liskov Substitution Principle



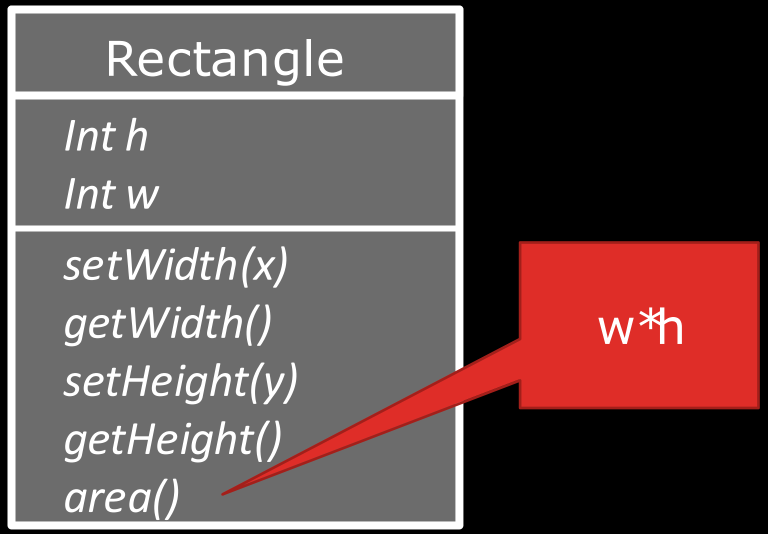
SOLID PRINCIPLES

Liskov Substitution Principle



SOLID PRINCIPLES

Liskov Substitution Principle



Rectangle

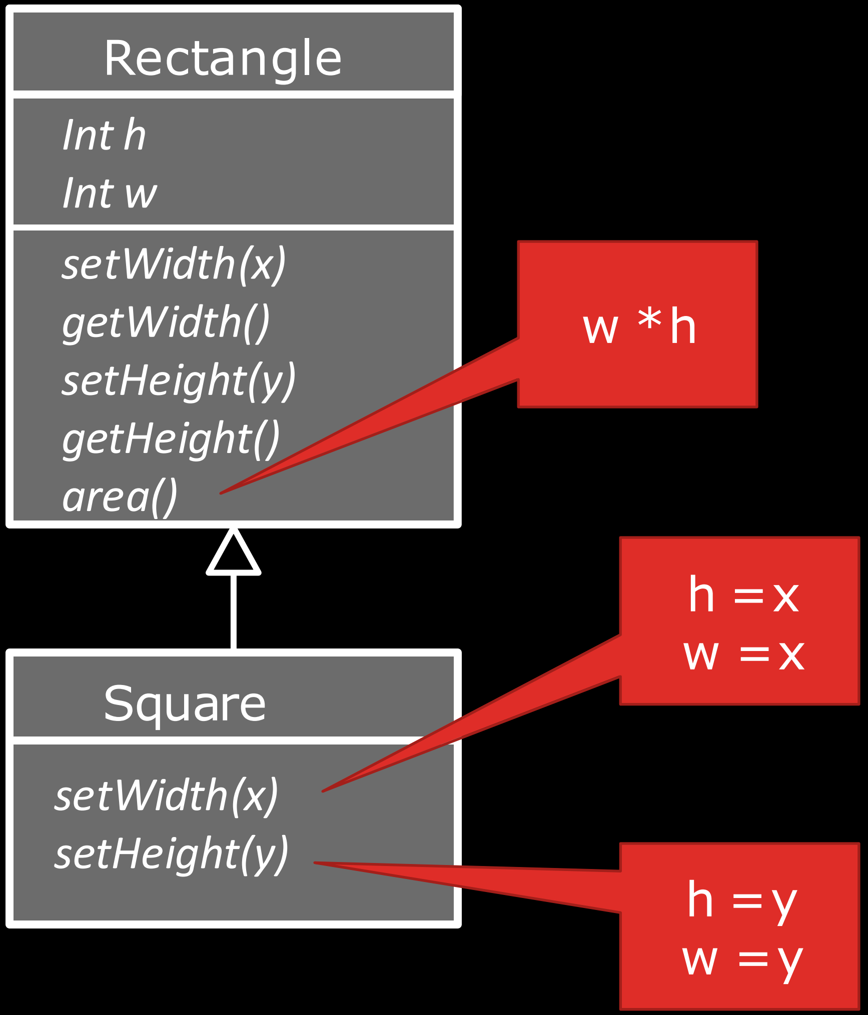
Int h
Int w

setWidth(x)
getWidth()
setHeight(y)
getHeight()
area()

w*h

SOLID PRINCIPLES

Liskov Substitution Principle

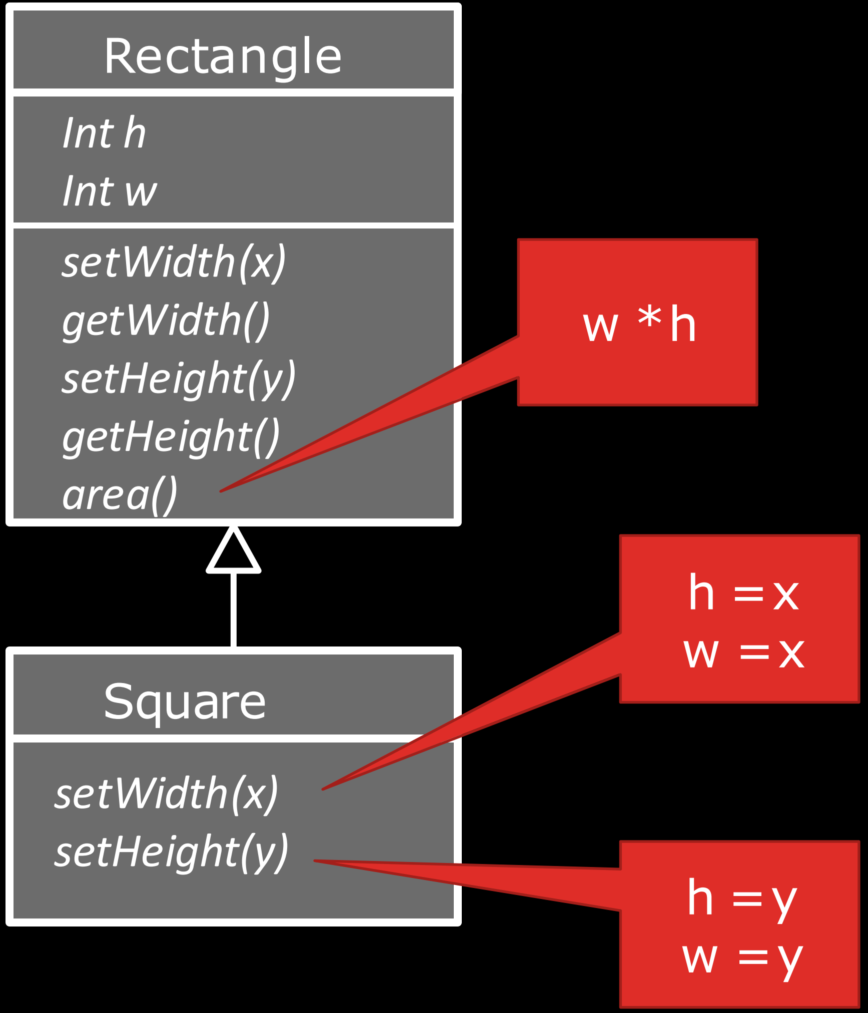


SOLID PRINCIPLES

Liskov Substitution Principle

Test method

```
shouldCalculateTheCorrectArea (Rectangle r, int width, int height)
 {

r.setWidth(width)
r.setHeight(height)

assertEquals(width * height, r.area ())



}
```



Rectangle

Int h
Int w

setWidth(x)
getWidth()
setHeight(y)
getHeight()
area()

w * h

Square

setWidth(x)
setHeight(y)

h = x
w = x

h = y
w = y

Interface Segregation Principle

SOLID PRINCIPLES

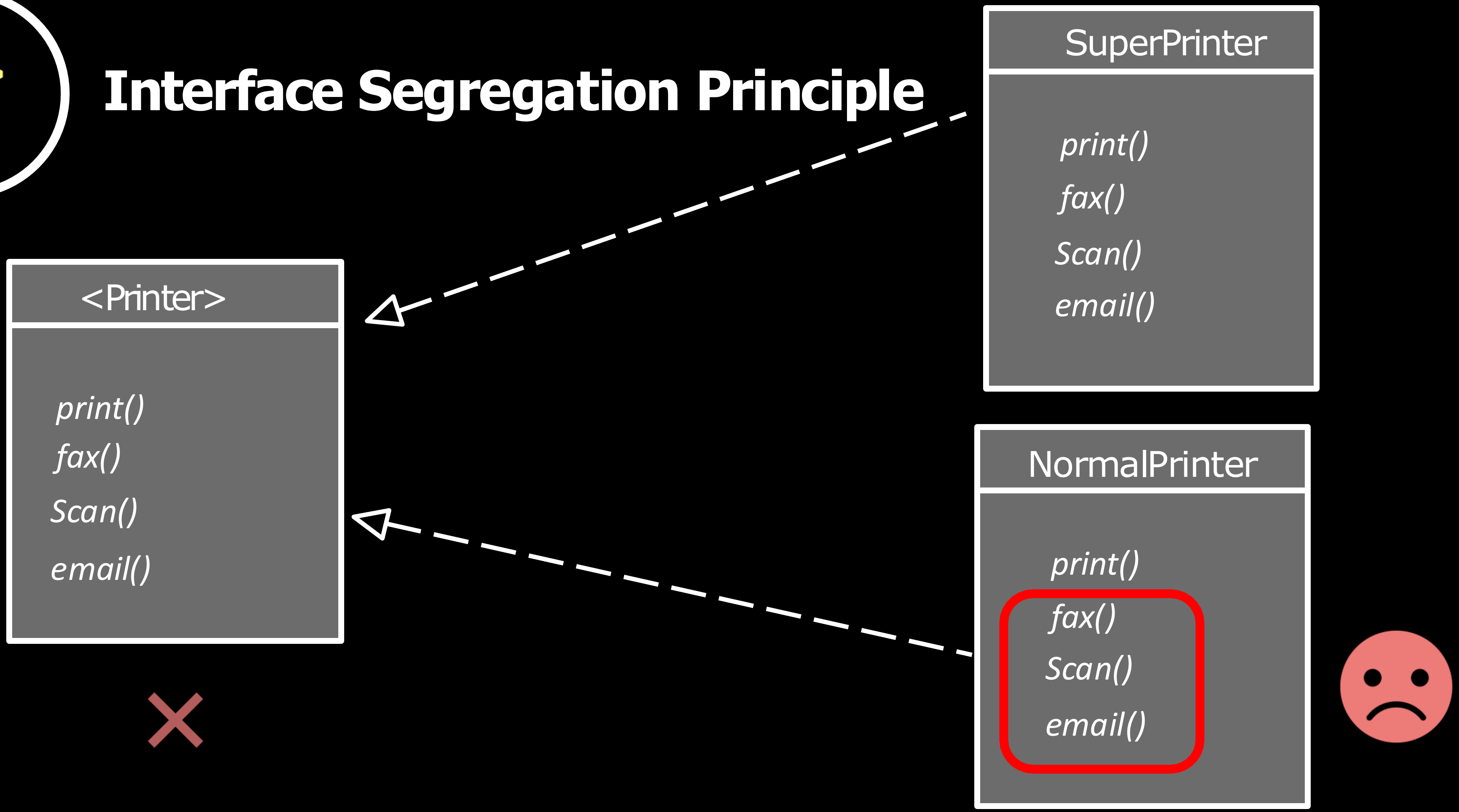
Interface Segregation Principle

"Many client-specific interfaces are better than one general-purpose interface."
Robert C. Martin

Classes should not be forced to depend on methods that they do not use

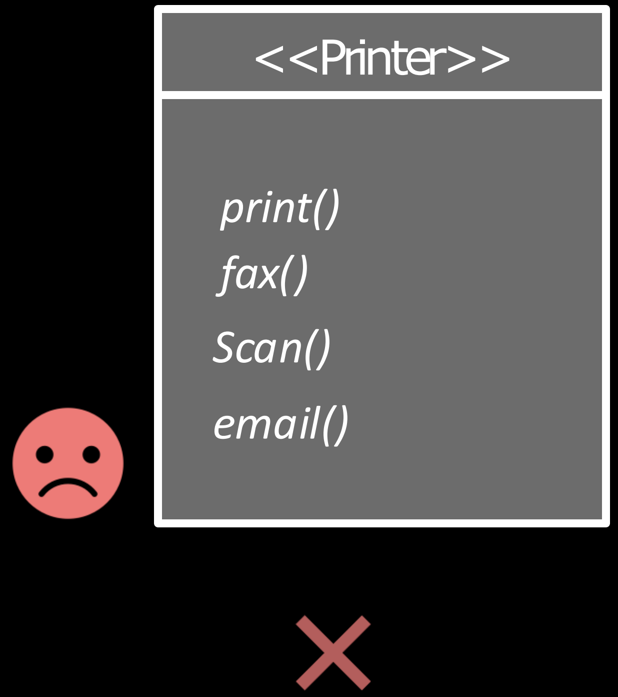
SOLID PRINCIPLES

Interface Segregation Principle



SOLID PRINCIPLES

Interface Segregation Principle

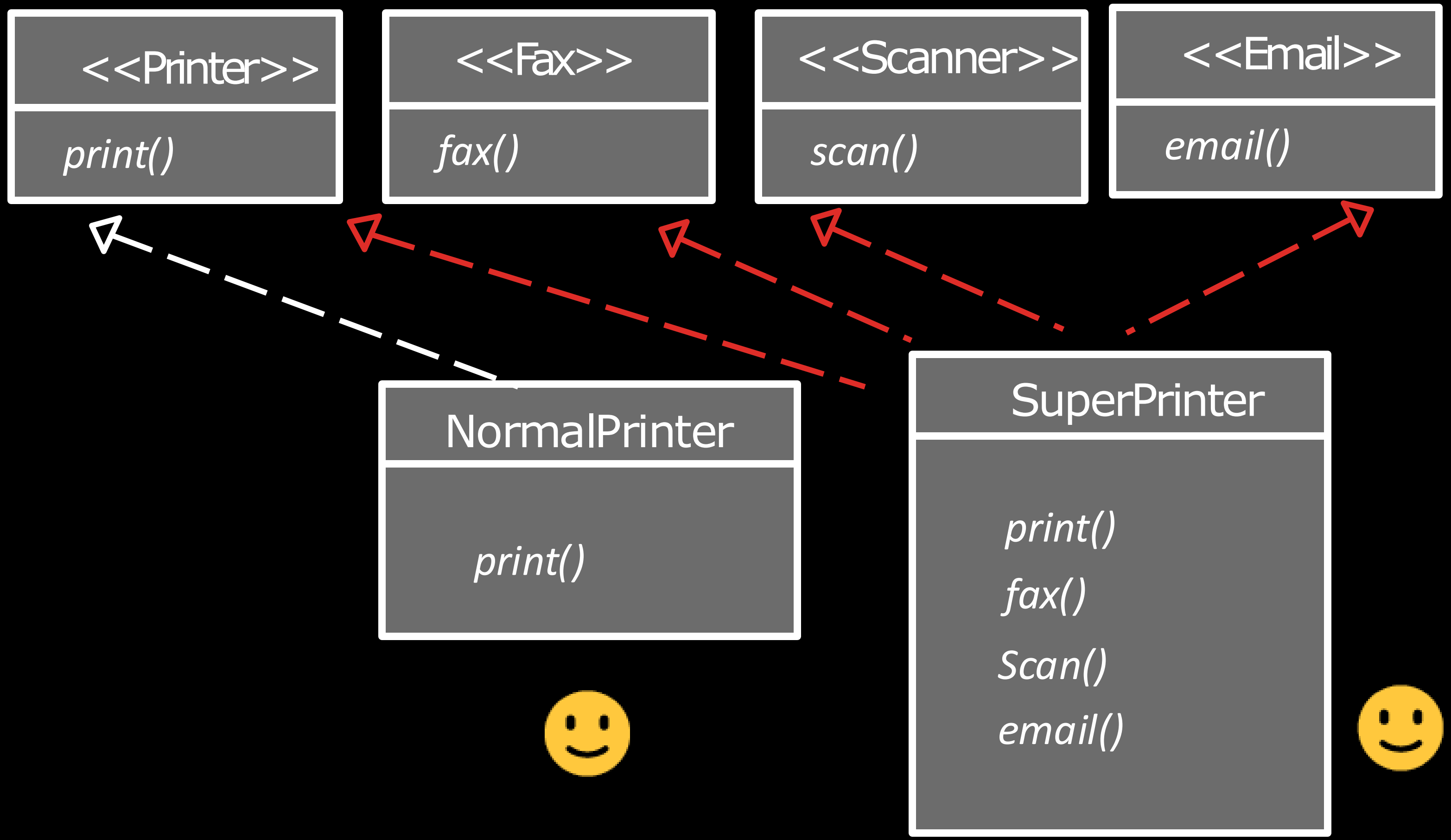


<<Printer>>

print()
fax()
Scan()
email()

✗

✓



<<Printer>>

print()

<<Fax>>

fax()

<<Scanner>>

scan()

<<Email>>

email()

NormalPrinter

print()

SuperPrinter

print()
fax()
Scan()
email()

Dependency Inversion Principle

SOLID PRINCIPLES

Dependency Inversion Principle

"One should depend upon abstractions, [not] concretions."
Robert C. Martin

High-level modules should not depend on low-level modules.

SOLID PRINCIPLES

Dependency Inversion Principle

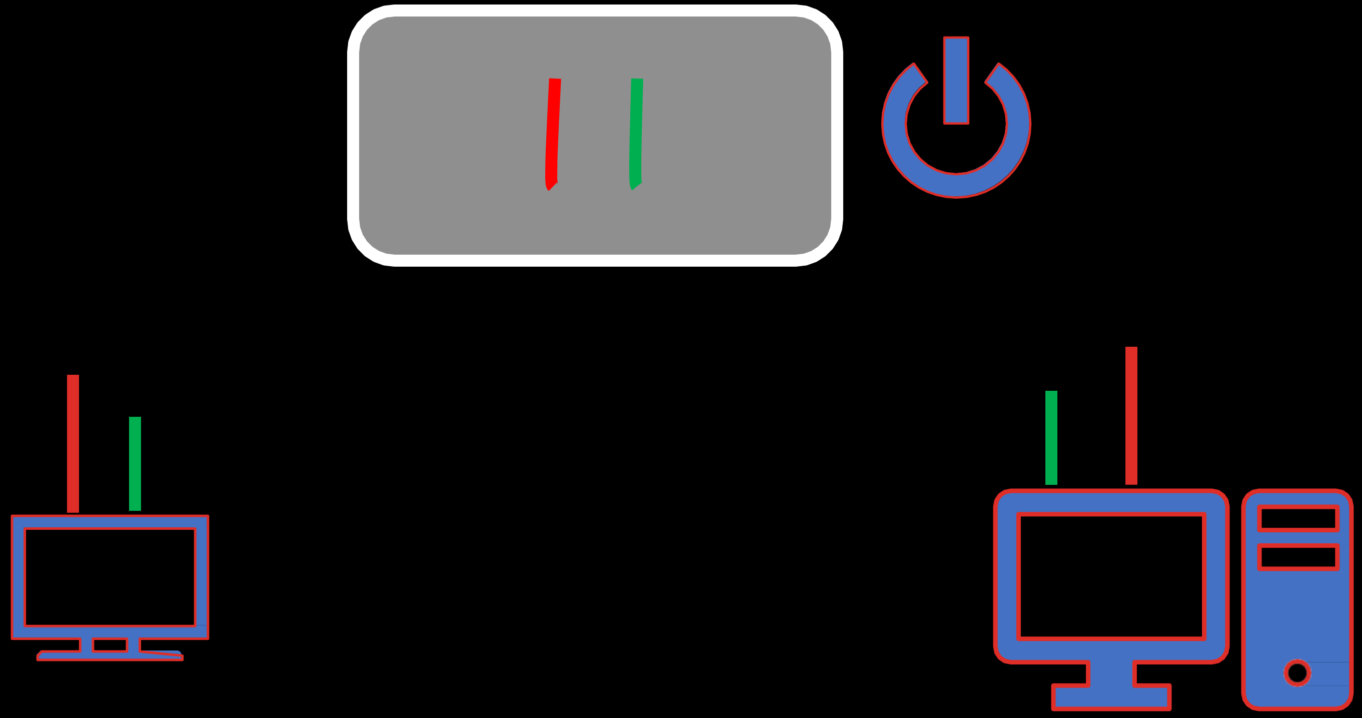
The dependency inversion principle (DIP) has two parts:

1. High-level modules should not depend on low-level modules. Instead, both should depend on abstractions (interfaces)

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

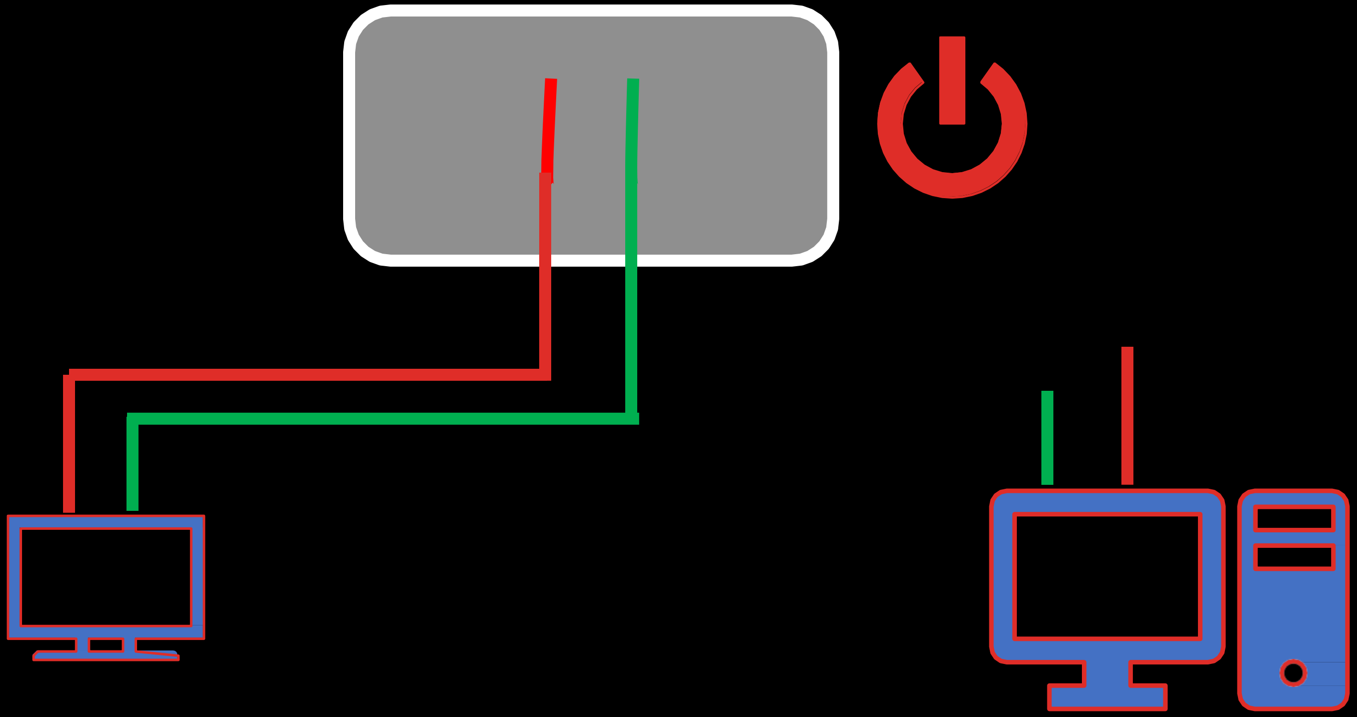
SOLID PRINCIPLES

Dependency Inversion Principle



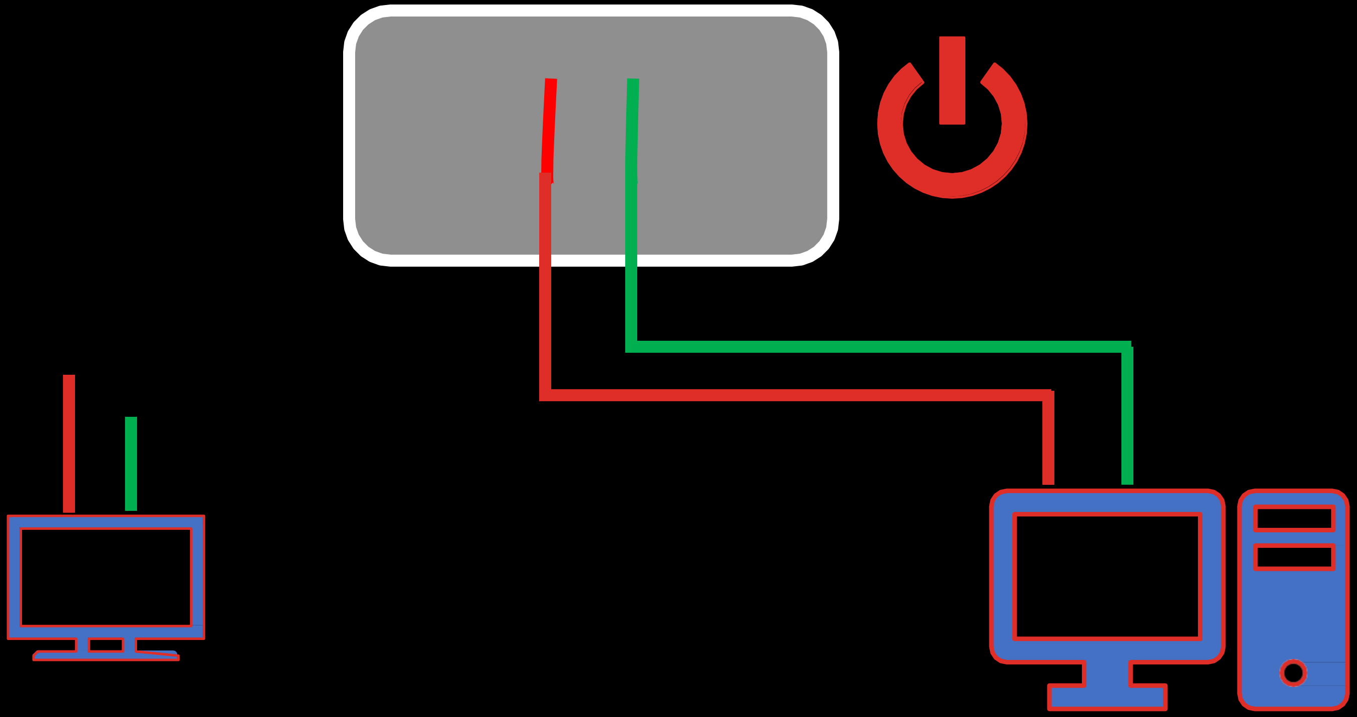
SOLID PRINCIPLES

Dependency Inversion Principle



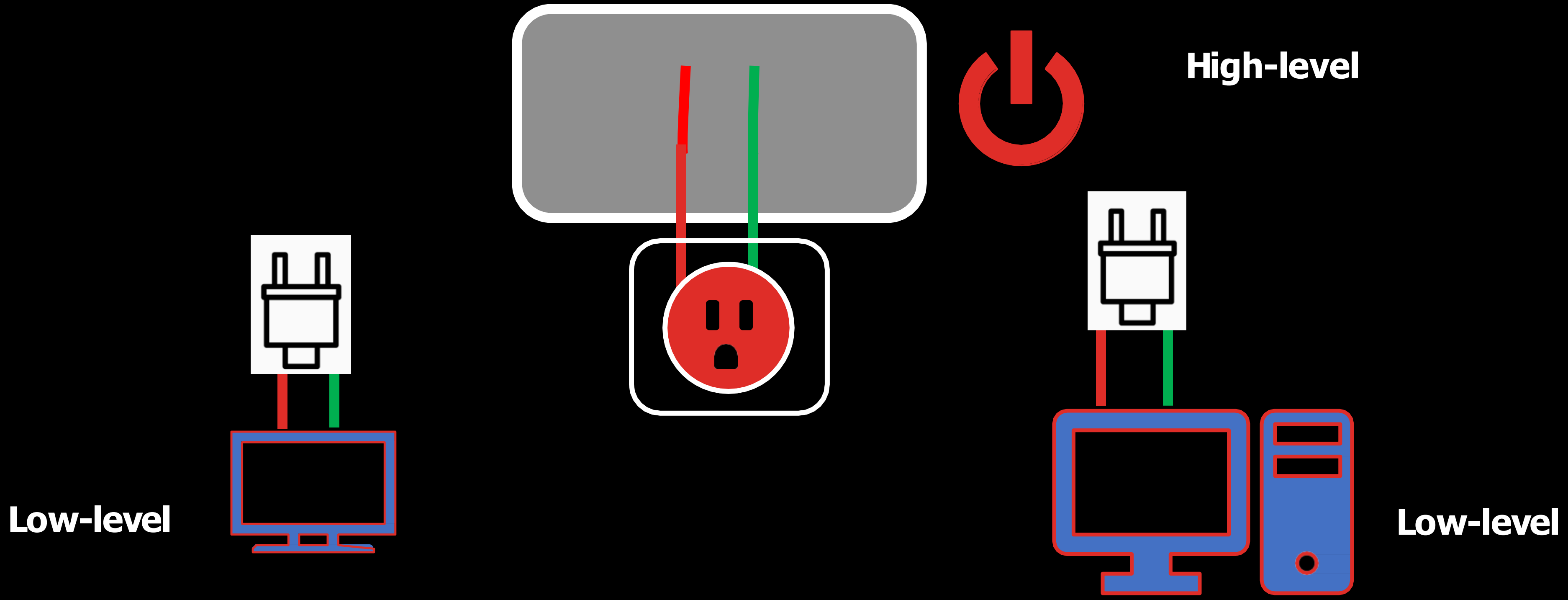
SOLID PRINCIPLES

Dependency Inversion Principle



SOLID PRINCIPLES

Dependency Inversion Principle



SOLID PRINCIPLES

Dependency Inversion Principle

Switch

☹

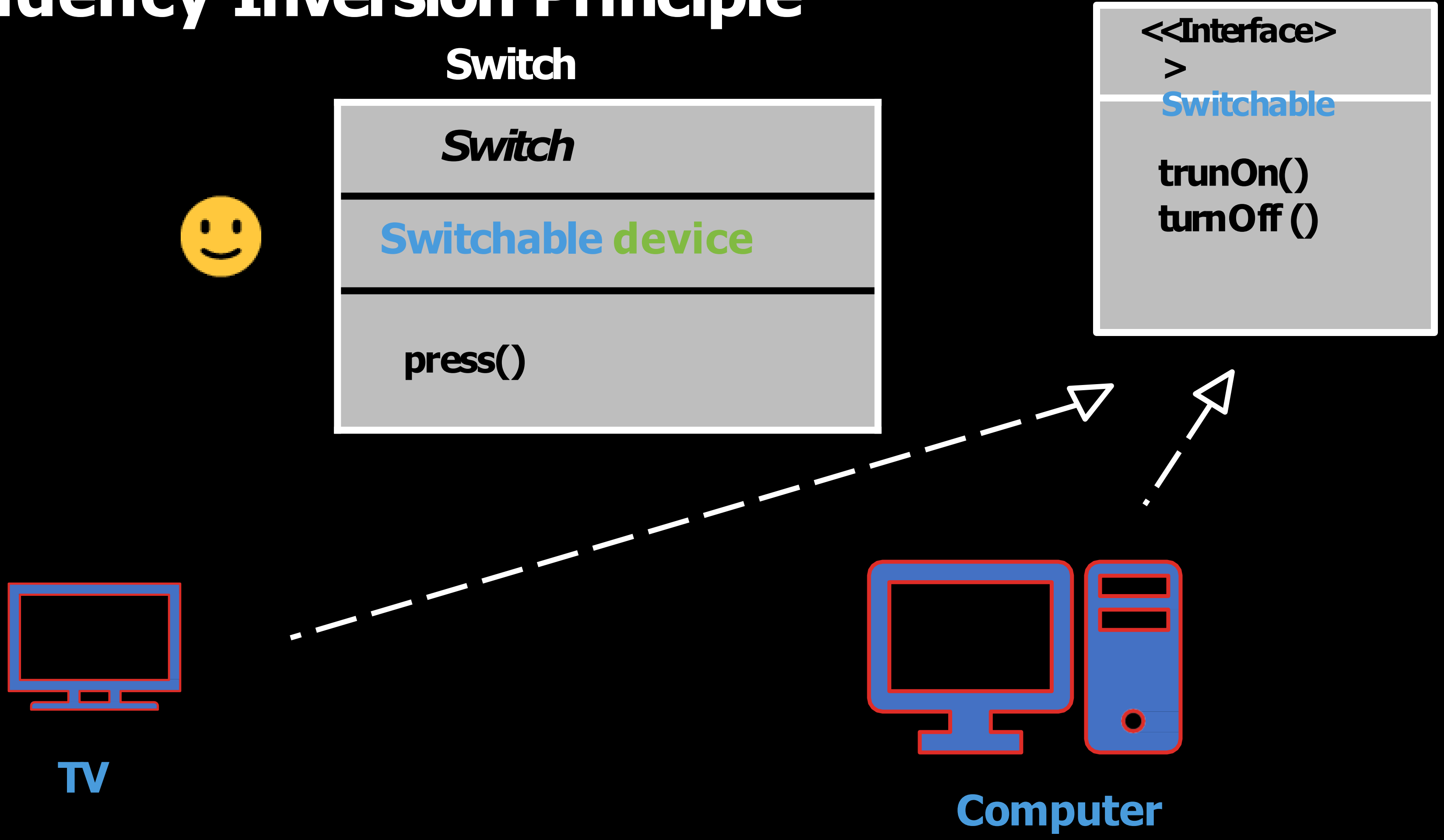
Switch
TV tv
press()

TV

Computer

SOLID PRINCIPLES

Dependency Inversion Principle



Switch

Switch

Switchable device

press()

<<Interface>
>
Switchable

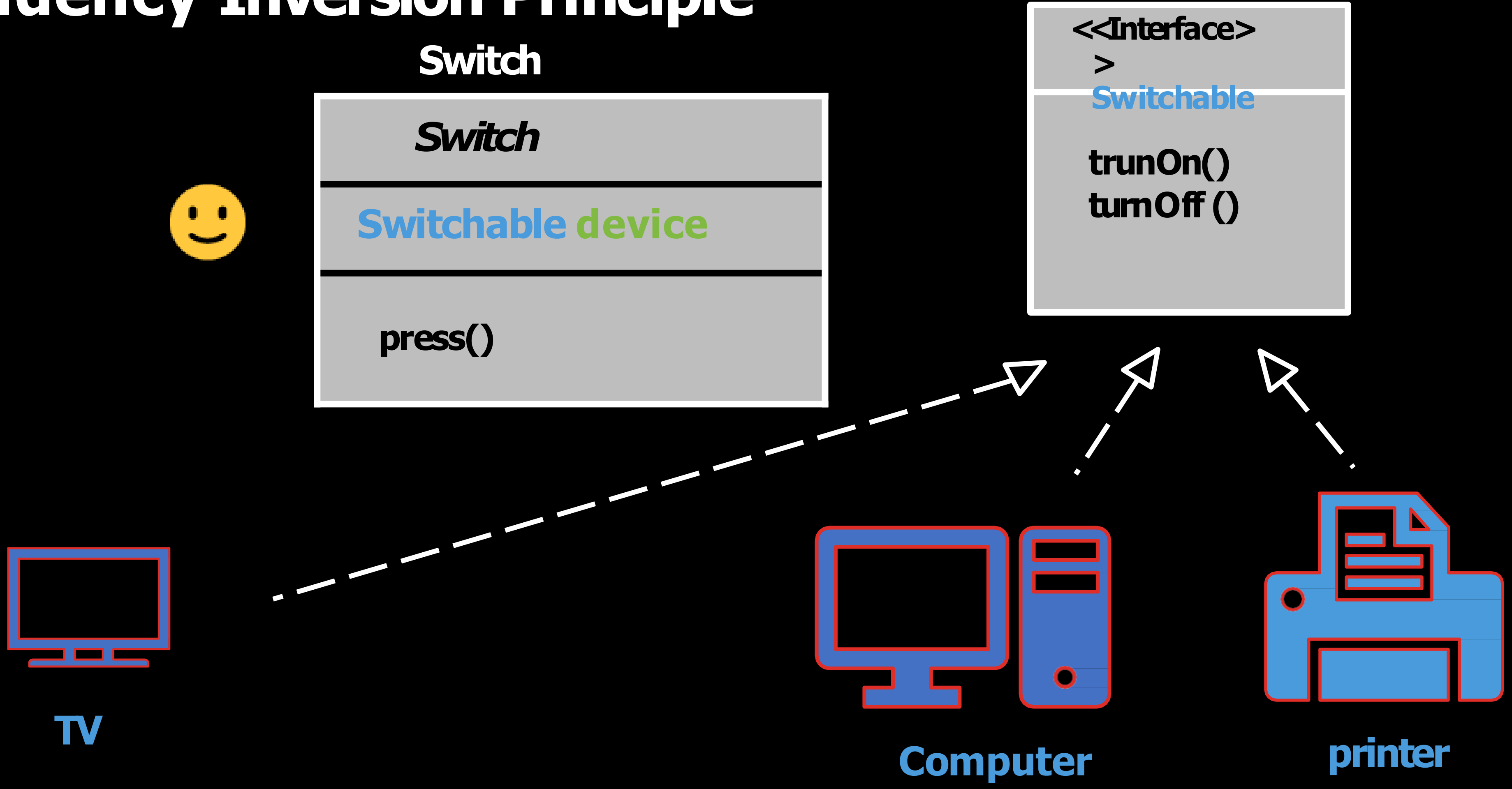
trunOn()
turnOff ()

TV

Computer

SOLID PRINCIPLES

Dependency Inversion Principle



Switch

Switch

Switchable device

press()

<<Interface>
>
Switchable

trunOn()
turnOff ()

TV

Computer

printer

SOLID PRINCIPLES

Summary

- The SOLID principles are an excellent way to improve your code and make modifications easier.
- It can be difficult to achieve them all in a single program if you’re just starting out, so first focus on one at a time.
- Eventually, you’ll write SOLID programs by habit ☺

NEXT STEPS

To do

Further Reading

FURTHER READING

Will be posted in the course shell

THANK YOU