Java Training
SOLID

Agenda

- Homework review
- SOLID
- Dependency injection
- Mockito
- homework

SOLID Principles

Acronym for the Five Principles of Object-Oriented Design

SOLID Principles

These principles, when applied together, are designed to increase the likelihood that a programmer will create a system that is easy to maintain and expand over time.

The SOLID Principles are guidelines that can be applied while working on software to remove "odor code" by instructing the programmer to refactor the source code until it is legible and extensible.

SOLID Principles

S – Single responsibility

O – Open/Closed

L – Liskov substitution

I – Interface segregation

D – Dependency inversion

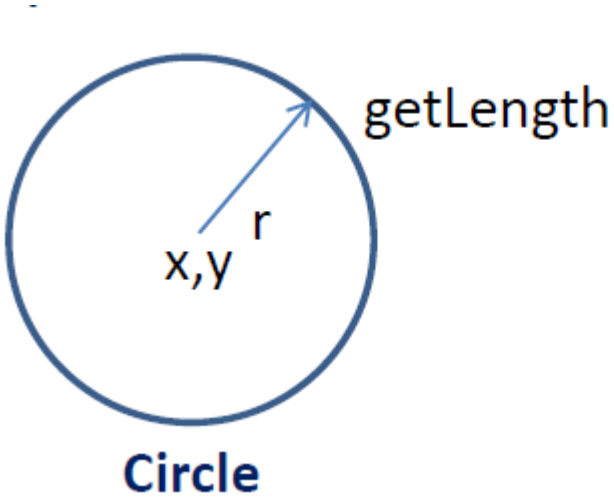
Single responsibility principle

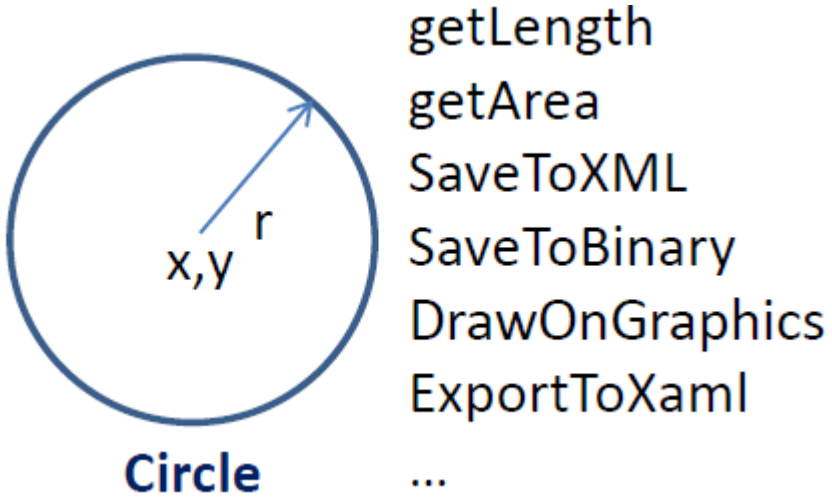
There should never be more than one reason for a class to change

In other words, every class should have only one responsibility

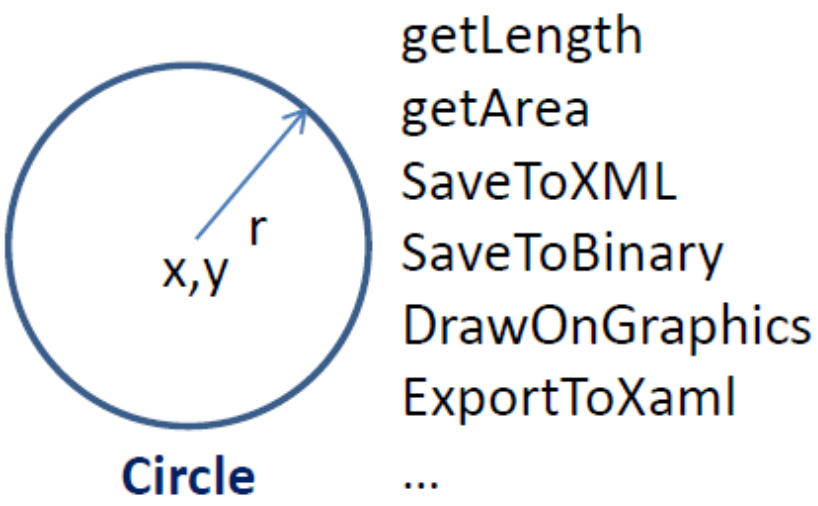
Single responsibility principle

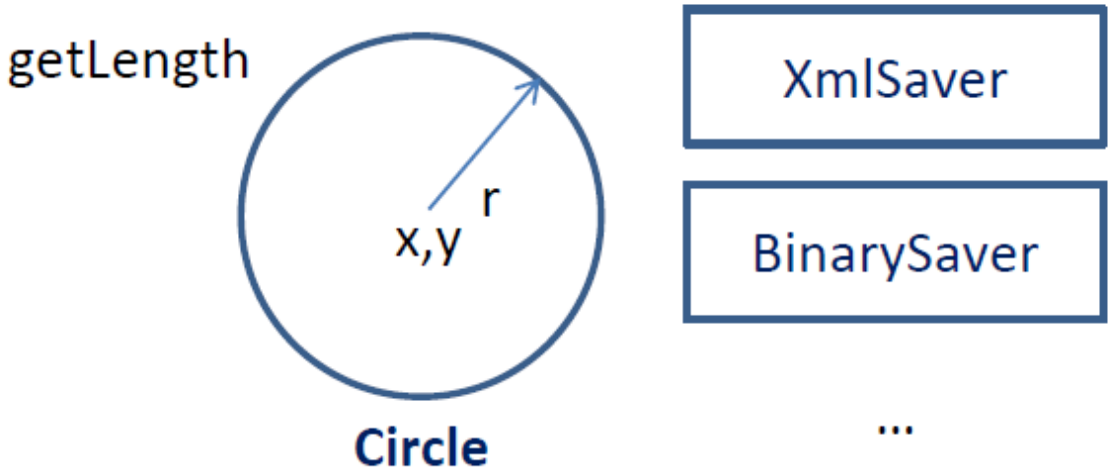
Single responsibility principle





Single responsibility principle





Open/Closed Principle

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

- open for extension: entity behavior can be extended by creating new entity types
- closed for modification: As a result of extending the behavior of an entity, no changes should be made to the code that these entities use.

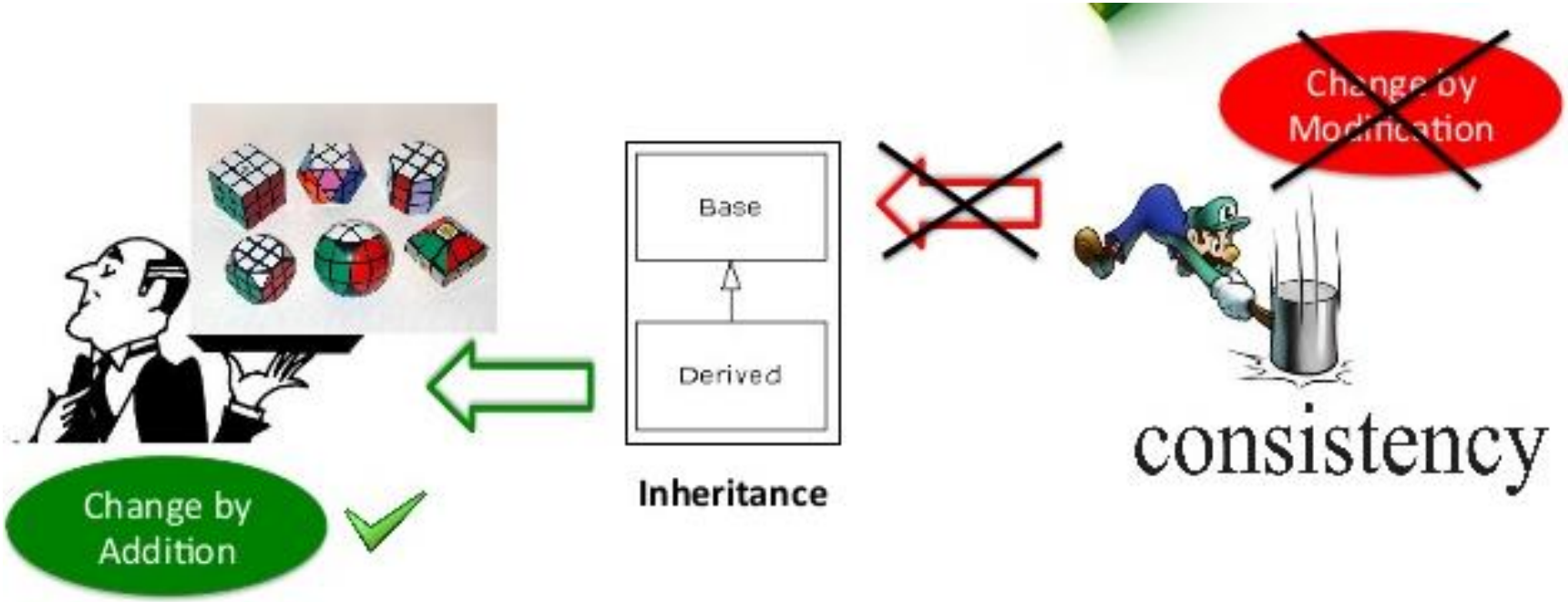
Open/Closed Principle

"How can you develop a project that is resistant to changes, the life of which exceeds the lifetime of the first version of the project?"

Bertrand Meyer

The idea was that once developed implementation of a class in the future requires only bug fixes, and new or changed functions require the creation of a new class. This new class can reuse the code of the original class through the inheritance mechanism. The derived subclass may or may not implement the interface of the original class.

Open/Closed Principle



Liskov substitution

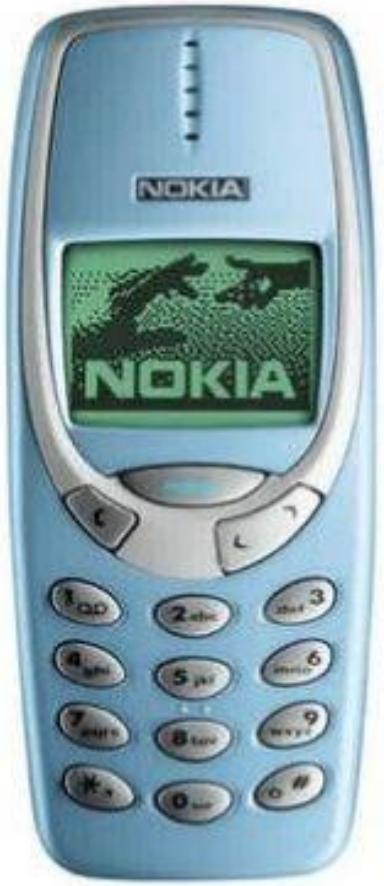
Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it

Liskov substitution

The behavior of inherited classes should not conflict with the behavior specified by the base class, that is, the behavior of inherited classes should be expected for code that uses a variable of the base type.

This principle warns the developer that changing behavior inherited from a derived type is very risky.

Liskov substitution



It is very important to follow this principle when designing new types using inheritance.

Liskov substitution

Interface segregation principle

Variant 1: Many client-specific interfaces are better than one general-purpose interface

Variant 2**:** clients shouldn't depend on methods they don't use

As with other principles of class design, we try to get rid of unnecessary dependencies in the code, make the code easy to read and easily changeable.

Interface segregation principle

```
interface Item {
    public function applyDiscount($discount);
    public function applyPromocode($promocode);
    public function setColor($color);
    public function setSize($size);
    public function setCondition($condition);
    public function setPrice($price);
    public function setMaterial($material);
}
```

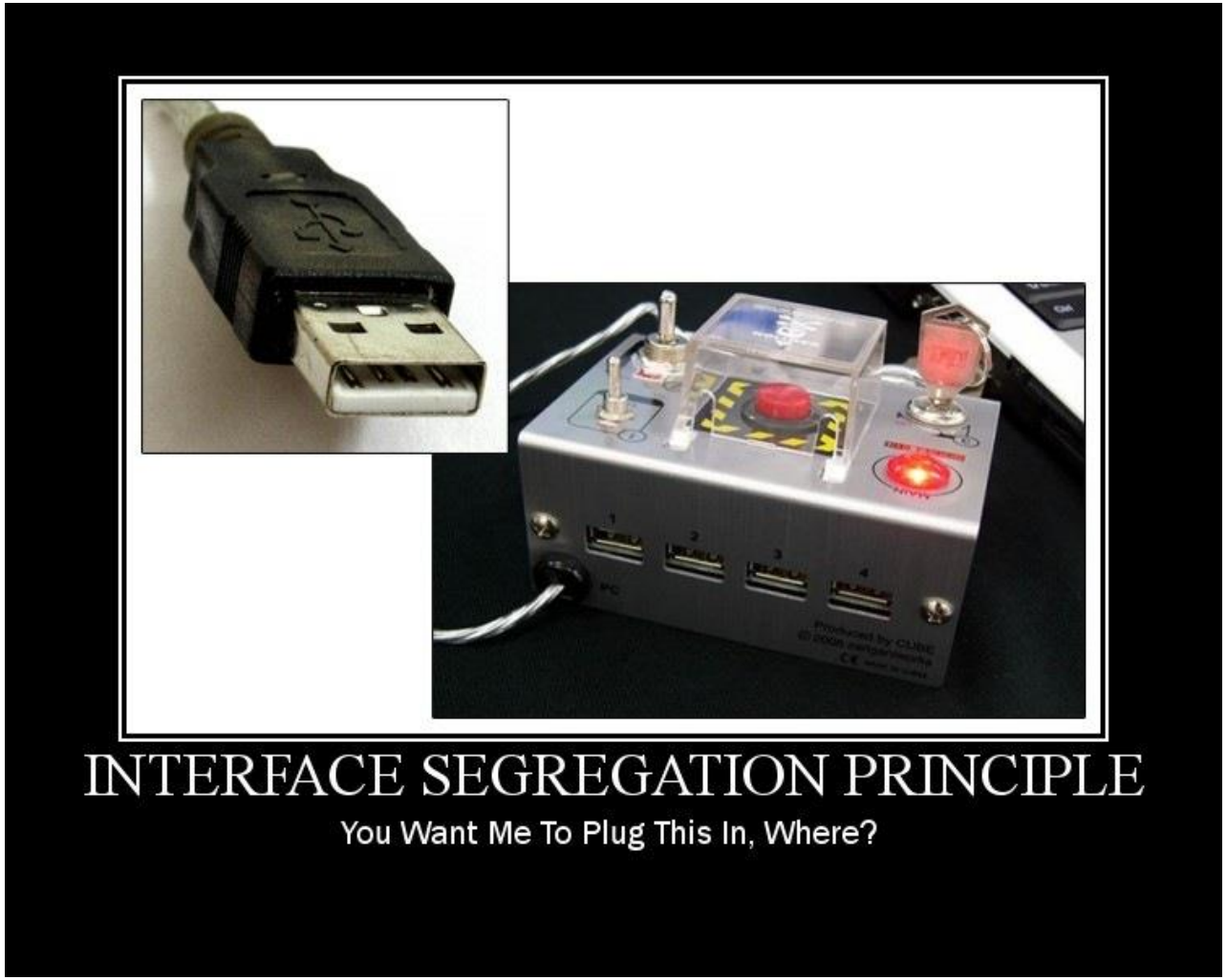
Interface segregation principle

```
interface Item {
    public function setCondition($condition);
    public function setPrice($price);
}

interface Clothes {
    public function setColor($color);
    public function setSize($size);
    public function setMaterial($material);
}

interface Discountable {
    public function applyDiscount($discount);
    public function applyPromocode($promocode);
}
```

Interface segregation principle

```
class Book implemets Item, Discountable {
    public function setCondition($condition){/*...*/}
    public function setPrice($price){/*...*/}
    public function applyDiscount($discount){/*...*/}
    public function applyPromocode($promocode){/*...*/}
}

class KidsClothes implemets Item, Clothes {
    public function setCondition($condition){/*...*/}
    public function setPrice($price){/*...*/}
    public function setColor($color){/*...*/}
    public function setSize($size){/*...*/}
    public function setMaterial($material){/*...*/}
}
```

Interface segregation principle

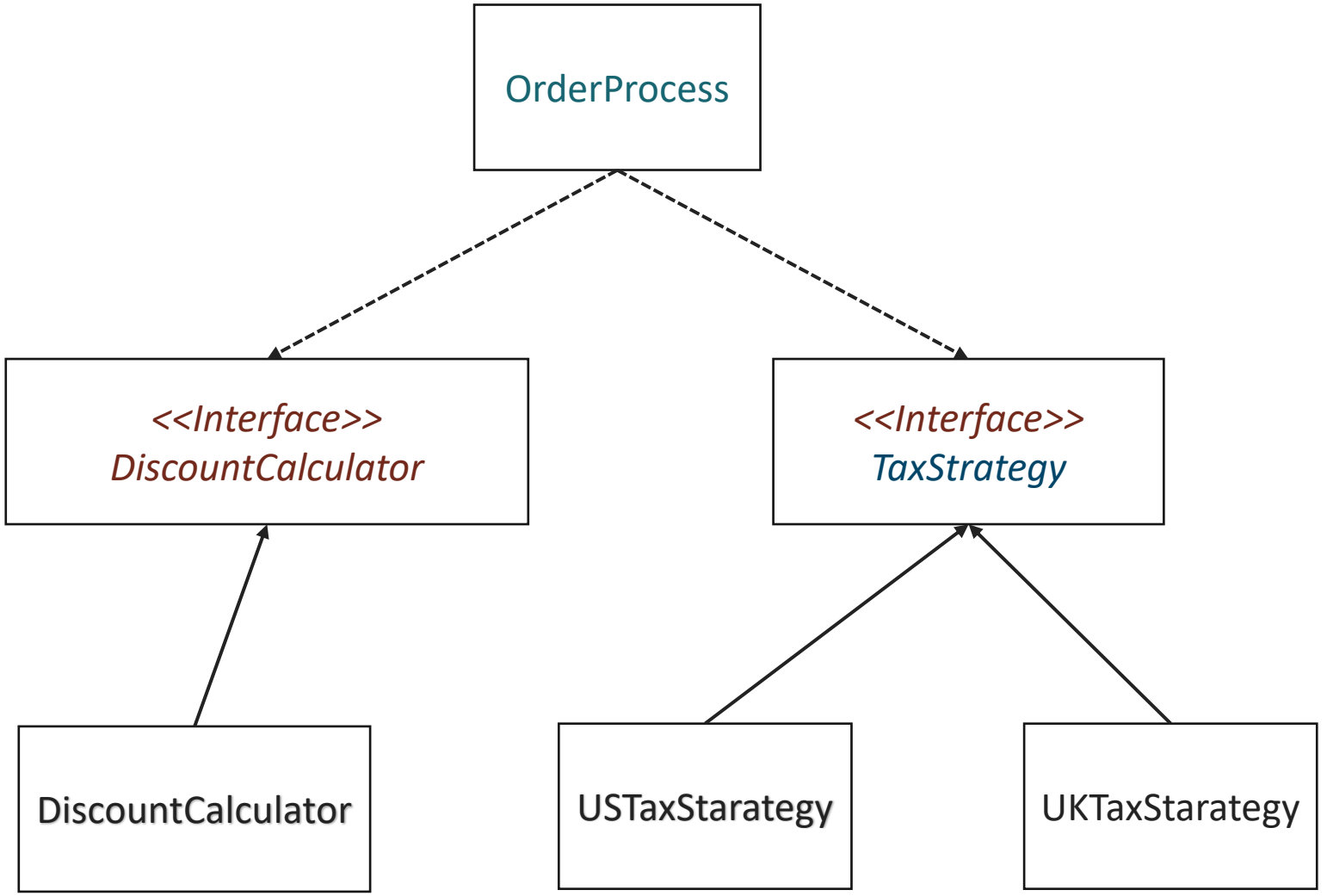


Dependency inversion principle

Definition:

- Top-level modules should not depend on lower-level modules. Both should depend on abstraction
- Depend upon abstractions, not implementation

Dependency inversion principle



```
class TaxStrategy {
        calculate(){
                ..
                If (country == UK){
                ...
                }
                ..
        }
}
```

```
class TaxStrategy {

        calculate(){
                ..
                doCountrySpecific()
                ..
         }

        doCountrySpecific() { }
}

class UKTaxStrategy extends TaxStrategy {

        overrride doCountrySpecific(){
        ...
        }

}
```

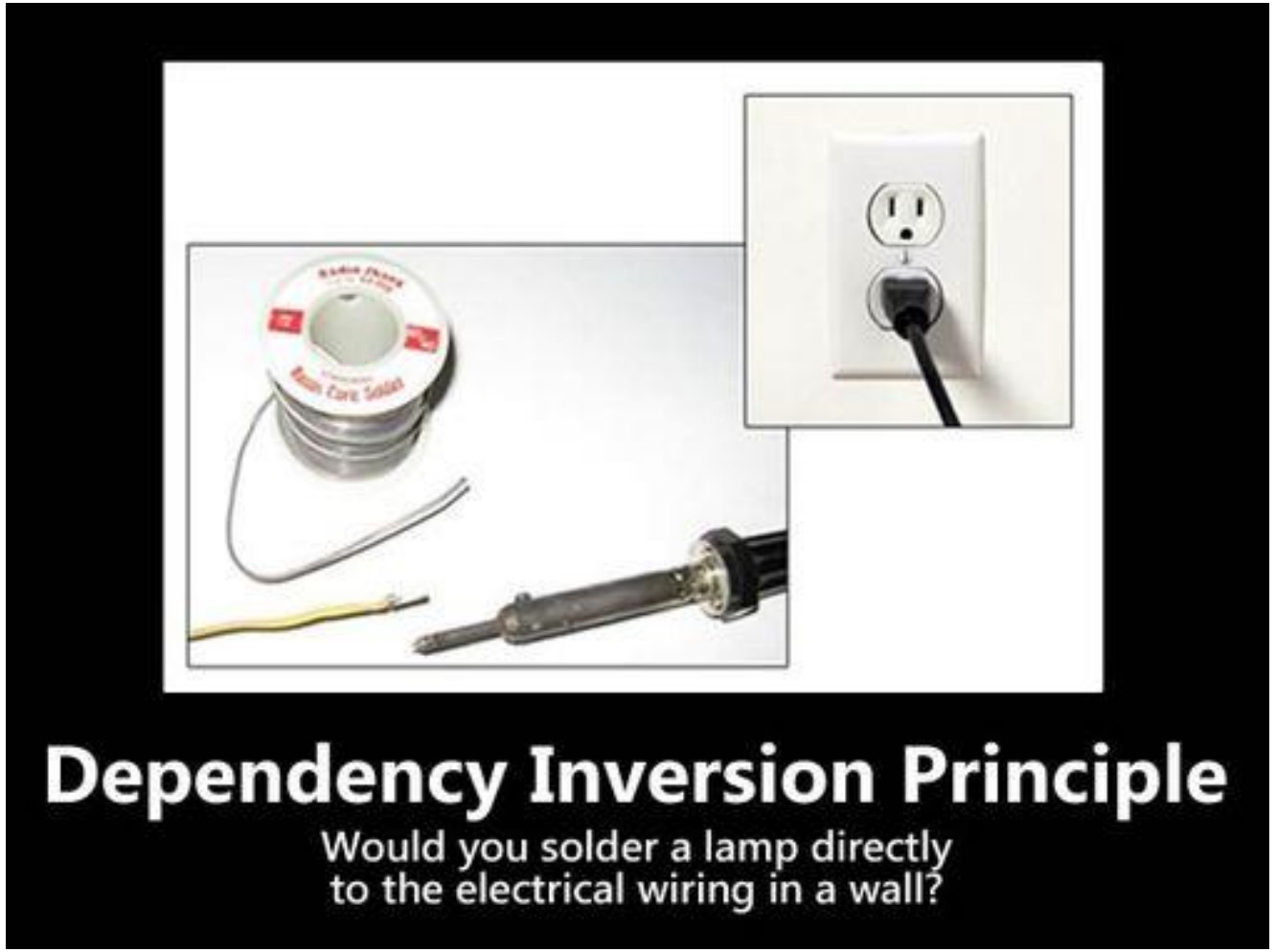
Dependency inversion principle

The reason projects get old is that developers don't have the ability to painlessly change the code of some components without fear of disrupting the work of others.

The design of such systems can be characterized by the following features:

- Rigidity - changing one part of the code affects too many other parts;
- Fragility - even a minor change in the code can lead to completely unexpected problems;
- Immobility - No part of the application can be easily isolated and reused.

Dependency inversion principle



Dependency Injection (DI)

Dependency injection

```
public class DataService {

    private final DataReader reader;


    public DataService(DataReader reader) {
        this.reader = reader;

    }

    public void process(String filename){
        String data = reader.readAll(filename);
        ...
    }
}
```

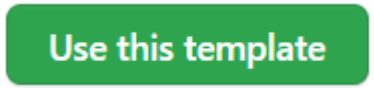
Mockito

```
DataReader reader = Mockito.mock(reader);

Mockito.when(reader.readAll(“input.txt")) .thenReturn(“Input line");

DataService service = new DataService(reader);
```

Homework

Circle. *Create the classes Point and Circle. Create methods and tests: calculate the area and perimeter of a circle; Is the shape a circle? (the radius cannot be <= 0); Does the figure intersects one of the coordinate axes at a specified distance?"*

Homework

- Fork from repo 

- To task according to description and project structure
- Write tests

https://github.com/filippstankevich/geometry

https://refactoring.guru/smells/long-method

Useful links

https://en.wikipedia.org/wiki/SOLID

https://medium.com/ibm-garage/solid-design-principles-makes-test-driven-development-faster-and-easier-35c9eec22ff1

Thanks