EECS3311 Fall 2022

Introduction. SOLID principles.

Ilir Dema

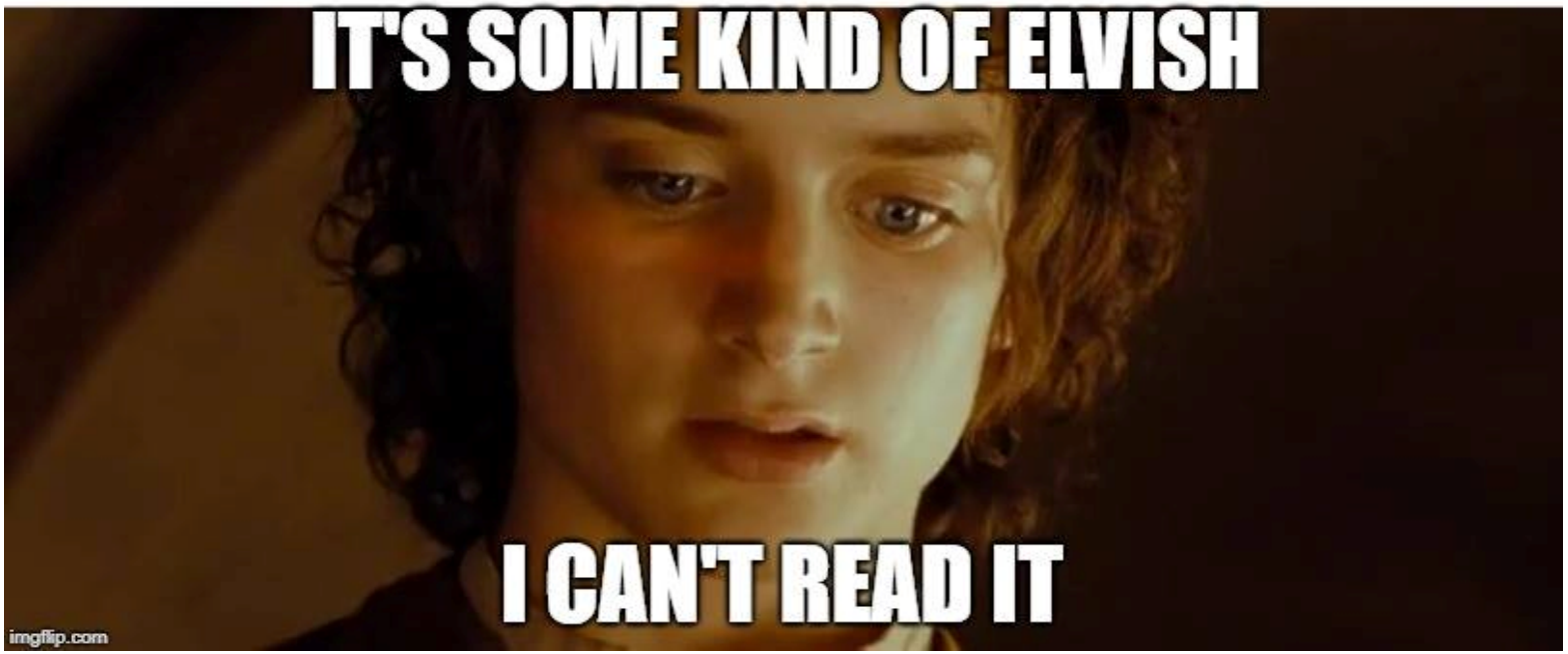
York University

Sep 7, 2022

Overview

WHY DO WE NEED SOFTWARE DESIGN?

When you trying to look at
the code you wrote a month ago



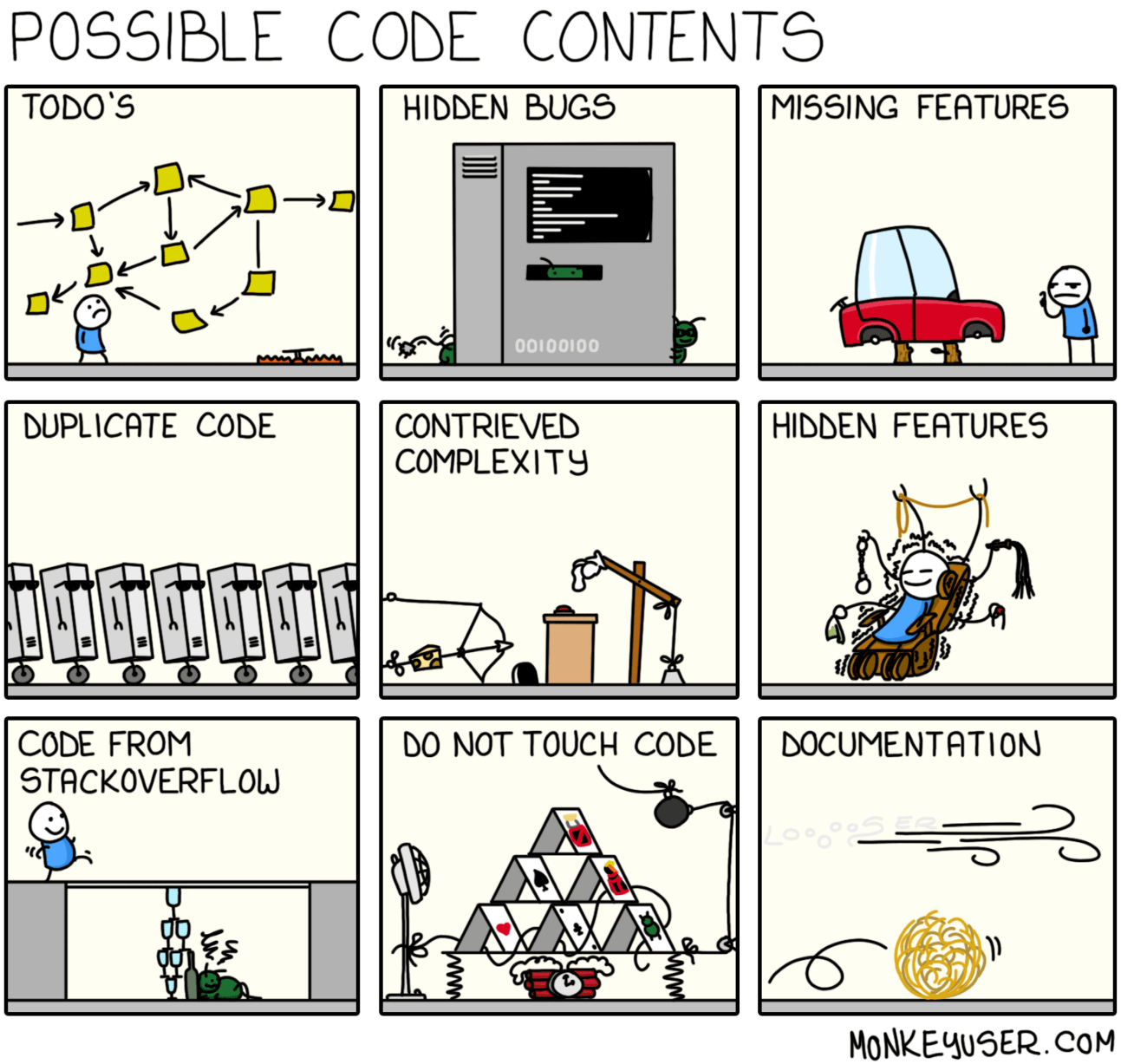
What is Software Design?

Software design is the process of envisioning and defining software solutions to one or more sets of problems.

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

Software design is basically a mechanism of preparing a plan, a layout for structuring the code of your software application.

IT'S ALL ABOUT HANDLING COMPLEXITY



IT'S ALL ABOUT HANDLING COMPLEXITY

It is All About Handling Complexity

- Techniques in our disposal:

1. Abstraction
   - Modeling and Object Orientation (encapsulation, information hiding)
2. Decomposition
   - Cohesion, Coupling, Modularity
3. Hierarchy
   - Specializations, Generalizations, Frameworks

IT'S ALL ABOUT HANDLING COMPLEXITY

Foundational Design Concepts

- ***Abstraction***: removal or masking of detail and complexity
- ***Modularity***: software is divided into separately named and addressable components
- ***Separation of Concerns***: any complex problem can be more easily handled if it is subdivided into pieces
- ***Information Hiding / Encapsulation***: hide implementation details from client components, communicate only via controlled interfaces
- ***High Cohesion***: components provide specific functionality
- ***Low Coupling***: components have as little as possible dependencies that is can change with minimal impact on other components

SOLID

- **S** Single responsibility principle
- **O** Open/closed principle
- **L** Liskov substitution principle
- **I** Interface segregation principle
- **D** Dependency inversion principle

A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE

S: Single Responsibility Principle

Definition: A class should have a single responsibility, where a responsibility is nothing but a reason to change.

- every class should have a single responsibility
- responsibility should be entirely encapsulated by the class
- all class services should be aligned with that responsibility

Why?

- makes the class more robust
- makes the class more reusable

SRP Example - Identify the problem

```
public class Employee{
  private String employeeId;
  private String name;
  private string address;
  private Date dateOfJoining;
  public boolean isPromotionDueThisYear(){
    //promotion logic implementation
  }
  public Double calcIncomeTaxForCurrentYear(){
    //income tax logic implementation
  }
  //Getters & Setters for all the private attributes
}
```

SRP Example - Issues

- The promotion logic is responsibility of HR.
- When promotion policies change, `Employee` class need not change.
- Similarly, tax computation is the finance department's responsibility.
- If `Employee` class owns the income tax calculation responsibility then whenever tax structure/calculations change Employee class will need to be changed.
- Lastly, `Employee` class should have the single responsibility of maintaining core attributes of an employee.

REFACTOR THE EMPLOYEE CLASS TO MAKE IT OWN A SINGLE RESPONSIBILITY.

Lets move the promotion determination logic from `Employee` class to the `HRPromotions` class:

```
public class HRPromotions{
  public boolean
       isPromotionDueThisYear(Employee emp){
    // promotion logic implementation
    // using the employee information passed
  }
}
```

REFACTOR THE EMPLOYEE CLASS TO MAKE IT OWN A SINGLE RESPONSIBILITY.

Similarly, lets move the income tax calculation logic from `Employee` class to `FinITCalculations` class:

```
public class FinITCalculations{
  public Double
        calcIncomeTaxForCurrentYear(Employee emp){
    //income tax logic implementation
    //using the employee information passed
  }
}
```

EMPLOYEE.JAVA ADHERING TO SINGLE RESPONSIBILITY PRINCIPLE.

Our `Employee` class now remains with a single responsibility of maintaining core employee attributes:

```
public class Employee{
   private String employeeId;
   private String name;
   private string address;
   private Date dateOfJoining;
   //Getters & Setters for all the private attributes
}
```

A CLASS SHOULD BE OPEN FOR EXTENSIBILITY, BUT CLOSED FOR MODIFICATION.

O: Open/Closed Principle

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
- add new features not by modifying the original class, but rather by extending it and adding new behaviours
- the derived class may or may not have the same interface as the original class
- originally defined by Bertrand Meyer in his book Object Oriented Software Construction.

O: Open/Closed Principle - 3 statements

1. A module will be said to be open if it is still available for extension. For example, it should be possible to new fields or new methods.
   - What it means: If attributes or behavior can be added to a class it can be said to be “open”.
2. A module will be said to be closed if it is available for use by other modules. This assumes that the module has been given a well-defined, stable description.
   - What it means: If a class is re-usable or specifically available for extending as a base class then it is closed.
3. A class adheres to the Open/Closed Principle when:
   - It is closed, since it may be compiled, stored in a library, baselined, and used by client classes
   - But it is also open, since any new class may use it as parent, adding new features.
   - What it means: A class can be open and closed at the same time.

Example of Open/Closed Principle in Java.

Lets say we need to calculate areas of various shapes. Say our first shape is `Rectangle`:

```
public class Rectangle{
 public double length;
 public double width;
}
```

Next we create a class to calculate area of this `Rectangle` which has a method `calculateRectangleArea()` :

```
public class AreaCalculator{
  public double calculateRectangleArea(Rectangle
    rectangle){
    return rectangle.length *rectangle.width;
  }
}
```

Example of Open/Closed Principle in Java.

Let's create a new class `Circle` with a single attribute `radius`:

```
public class Circle{
 public double radius;
}
```

Then we modify `AreaCalculator` class:

```
public class AreaCalculator{
  public double calculateRectangleArea(Rectangle
      rectangle){
    return rectangle.length *rectangle.width;
  }
  public double calculateCircleArea(Circle circle){
    return Math.PI*circle.radius*circle.radius;
  }
}
```

O: Open/Closed Principle - Identify the flaws

- Lets say we have a new shape pentagon next. In that case we will again end up modifying `AreaCalculator` class.
  - As the types of shapes grows this becomes messier as `AreaCalculator` keeps on changing and any consumers of this class will have to keep on updating their libraries which contain `AreaCalculator`
  - As a result, `AreaCalculator` class will not be baselined(finalized) with surety as every time a new shape comes it will be modified.
  - So, this design is not closed for modification.

O: Open/Closed Principle - Identify the flaws

- Also, note that this design is not extensible:
  - As we add more shapes, `AreaCalculator` will need to keep on adding their computation logic in newer methods.
  - We are not really expanding the scope of shapes; rather we are simply doing piece-meal(bit-by-bit) solution for every shape that is added.

MAKE THE DESIGN EXTENSIBLE.

For this we need to first define a base type `Shape`

```
public interface Shape{
  public double calculateArea();
}
```

MAKE THE DESIGN EXTENSIBLE.

Next, have `Circle & Rectangle` implement `Shape` interface

```
public class Rectangle implements Shape{
  double length;
  double width;
  public double calculateArea(){
    return length * width;
  }
}

public class Circle implements Shape{
  public double radius;
  public double calculateArea(){
    return Math.PI*radius*radius;
  }
}
```

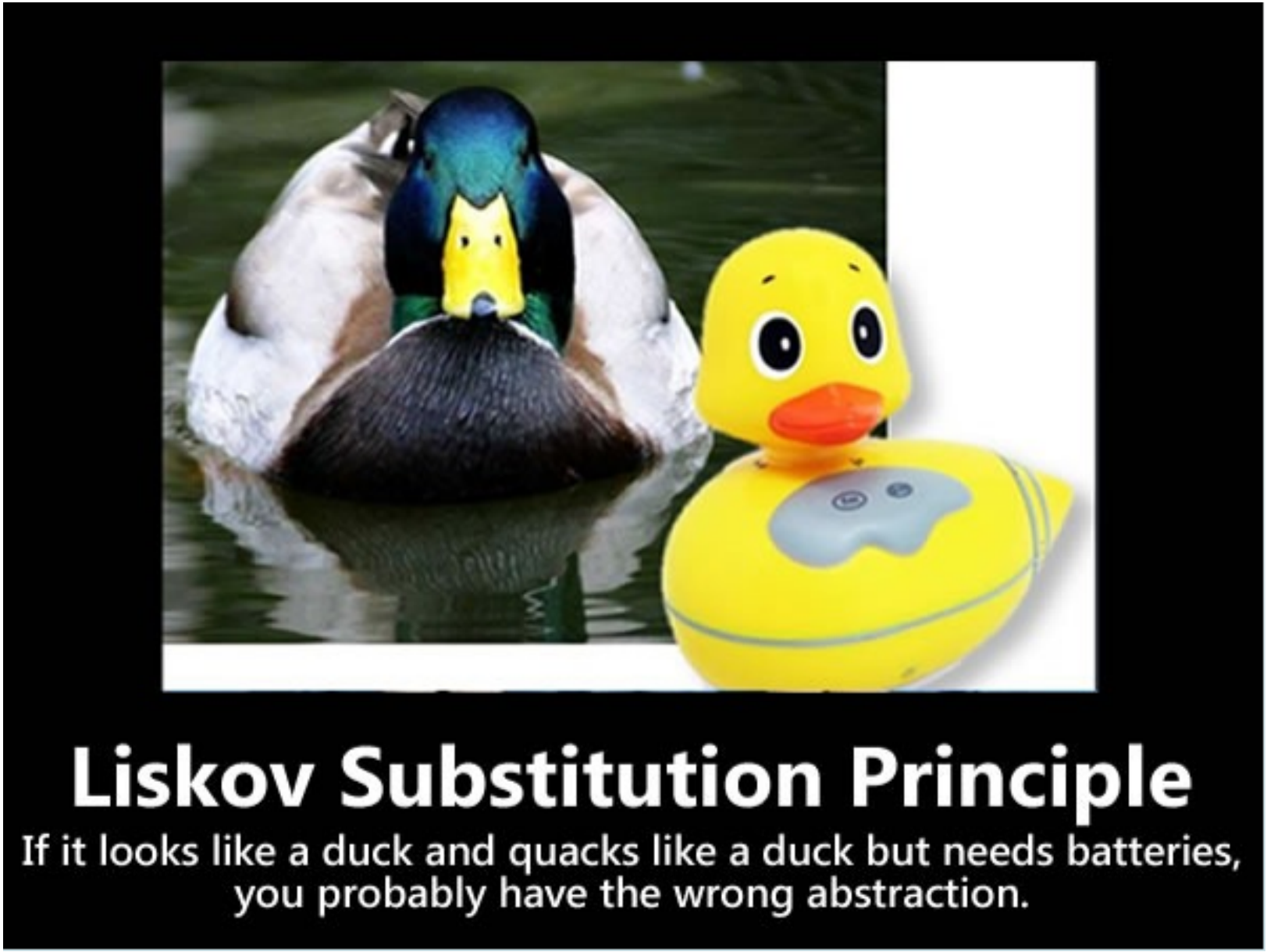
MAKE THE DESIGN CLOSED.

In our case consumer will be the `AreaCalculator` class which would now look like this

```
public class AreaCalculator{
  public double calculateShapeArea(Shape shape){
    return shape.calculateArea();
  }
}
```

This `AreaCalculator` class now fully removes our design flaws noted above and gives a clean solution which adheres to the Open-Closed Principle.

Liskov substitution principle

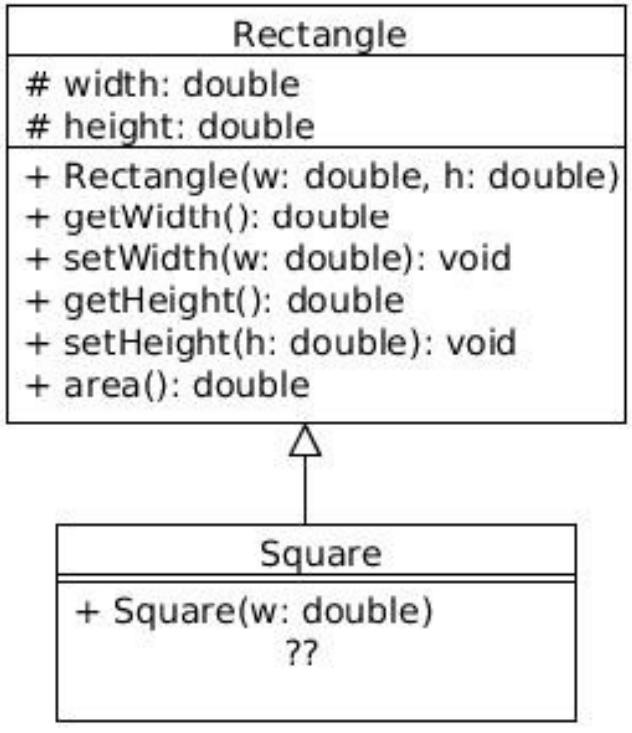


L: Liskov substitution principle

- If S is a subtype of T, then objects of type S may be substituted for objects of type T, without altering any of the desired properties of the program.
- “S is a subtype of T”? In Java, S is a child class of T, or S implements interface T.
- For example, if C is a child class of P, then we should be able to substitute C for P in our code without breaking it.

LISKOV SUBSTITUTION PRINCIPLE

- A classic example of breaking this principle:



Exercise

- Write (paper and pencil is fine) an implementation of `Rectangle`.
- Extend the `Rectangle` class to a `Square` class.
- Declare a `List<Rectangle>` and stuff in a number of `Rectangle` and `Square` objects.
- After all, each `Square` is a `Rectangle`, so we should have no problem doing that.
- Pass the list to a cutting machine that cuts wood shapes.
- An emergency requirement comes in: due to new packaging rules, no shape can have height greater than a preset `MAX_HEIGHT`.

EXERCISE (CONTINUED)

```
class CuttingMachine {
    // ... some code ...
    protected void fixHeight(List<Rectangle>
        rectangleList, double maxHeight) {
        for rectangle in rectangleList
            rectangle.setHeight(maxHeight);
    }
}
```

1. Explain what might happen.
2. Solve the problem.

L: Liskov substitution principle

- In OO programming and design, unlike in math, it is not the case that a Square is a Rectangle!
- This is because a Rectangle has more behaviours than a Square, not less.
- The LSP is related to the Open/Close principle: the sub classes should only extend (add behaviours), not modify or remove them.

CLIENTS SHOULD NOT BE FORCED TO DEPEND ON METHODS THEY DO NOT USE

I: Interface segregation principle

- No client should be forced to depend on methods it doesn't use.
- Better to have lots of small, specific interfaces than fewer larger ones.
- Easier to extend and modify the design.

EXAMPLE: THE RESTAURANT INTERFACE

```
public interface RestaurantInterface {
   public  void acceptOnlineOrder();
   public  void takeTelephoneOrder();
   public  void payOnline();
   public  void walkInCustomerOrder();
   public  void payInPerson();
}
```

OnlineClientImpl.java

```
public class OnlineClientImpl implements RestaurantInterface{
  @Override
  public void acceptOnlineOrder() {
    //logic for placing online order
  }
  @Override
  public void takeTelephoneOrder() {
    //Not Applicable for Online Order
    throw new UnsupportedOperationException();
  }
  @Override
  public void payOnline() {
    //logic for paying online
  }
  @Override
  public void walkInCustomerOrder() {
    //Not Applicable for Online Order
    throw new UnsupportedOperationException();
  }
  @Override
  public void payInPerson() {
    //Not Applicable for Online Order
    throw new UnsupportedOperationException();
  }
}
```

IDENTIFYING PROBLEMS ...

- Since `OnlineClientImpl.java` is for online orders, the other methods throw `UnsupportedOperationException`.
- Since the 5 methods are part of the RestaurantInterface, the implementation classes have to implement all 5 of them.
- Implementing all methods is inefficient.
- Any change in any of the methods of the RestaurantInterface will be propagated to all implementation classes. Maintenance of code then starts becoming really cumbersome and regression effects of changes will keep increasing.
- RestaurantInterface.java breaks Single Responsibility Principle because the logic for payments as well as that for order placement is grouped together in a single interface.

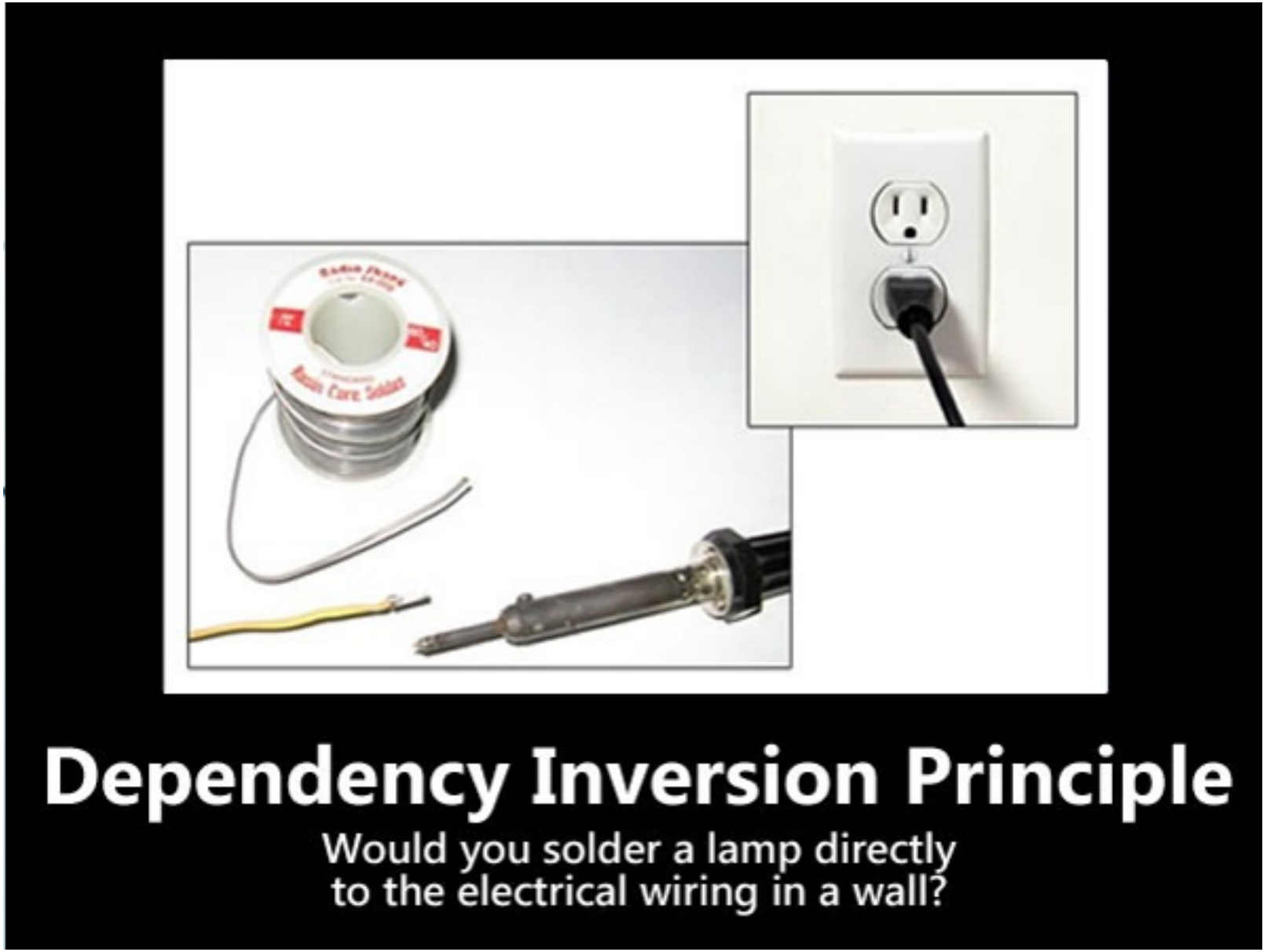
Applying Interface Segregation Principle

- Separate out payment and order placement functionalities into two separate lean interfaces. Let's name them PaymentInterface.java and OrderInterface.java.
- Each of the clients use one implementation each of PaymentInterface and OrderInterface. For example - OnlineClient.java uses OnlinePaymentImpl and OnlineOrderImpl and so on.
- Change in any one of the order or payment interfaces does not affect the other. They are independent now.
- There will be no need to do any dummy implementation or throw an UnsupportedOperationException as each interface has only methods it will always use.

D: Dependency inversion principle

- When building a complex system, we may be tempted to define the “low-level” classes first and then build the “higher-level” classes that use the low-level classes directly.
- (for simplicity, “low-level” - components, “high-level” - classes that use those components)
- But this approach is not flexible! What if we need to replace a low-level class? The logic in the high-level class will need to be replaced.
- To avoid such problems, we can introduce an abstraction layer between low-level classes and high -level classes.

Dependency inversion principle



Dependency Inversion Principle

- Depending on which source one refers to, Dependency Inversion Principle, as defined by Robert C. Martin, can be defined in any of the following ways:
  - Depend upon Abstractions. Do not depend upon concretions.
  - Abstractions should not depend upon details. Details should depend upon abstractions.
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.

Dependency Inversion Example

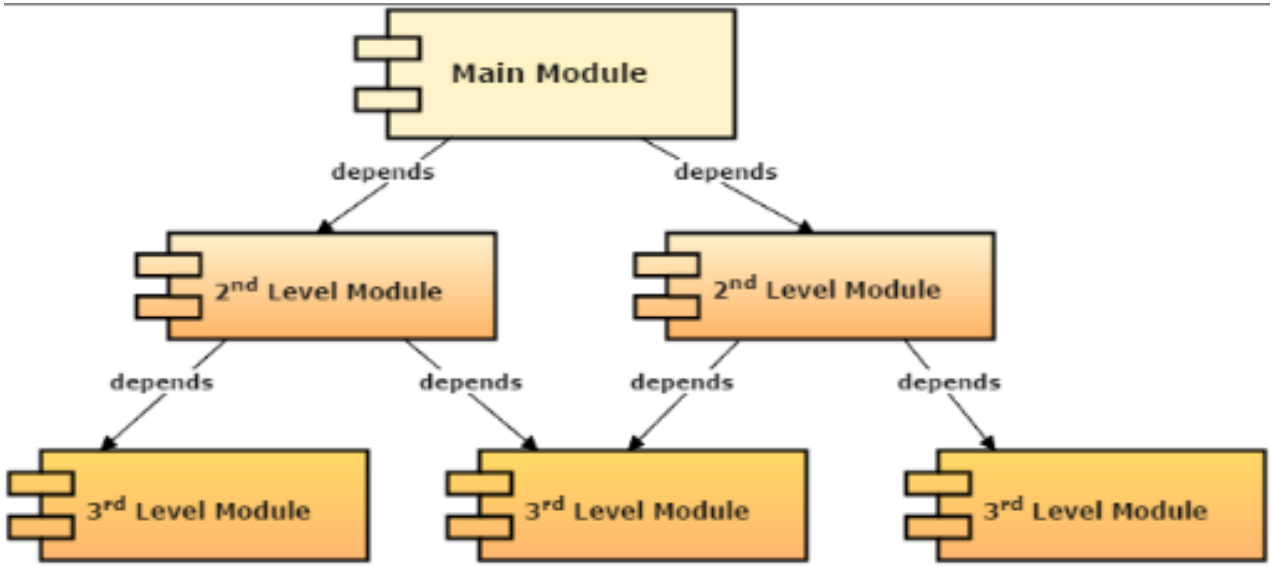
```
// Identify and fix the problem
public class DoBackEnd {// Low level module
    public void writeJava() {...}
}
public class DoFrontEnd {// Low level module
    public void writeJavascript() {...}
}
public class Project { // high level module
  private DoBackEnd backEnd= new DoBackEnd();
  private DoFrontEnd frontEnd = new DoFrontEnd();
  public void implement() {
        backEnd.writeJava();
        frontEnd.writeJavascript();
  }
}
```

Dependency Inversion Solution

```
public interface Developer {void develop();}
public class DoBackEnd implements Developer {
    @Override
    public void develop() {writeJava();}
    private void writeJava() {...}
}
public class DoFrontEnd {// Similar }
public class Project { // high level module
    private List<Developer> developers;
    public Project(List<Developer> developers) {
        this.developers = developers;
    }
    public void implement() {
        developers.forEach(d->d.develop());
    }
}
```

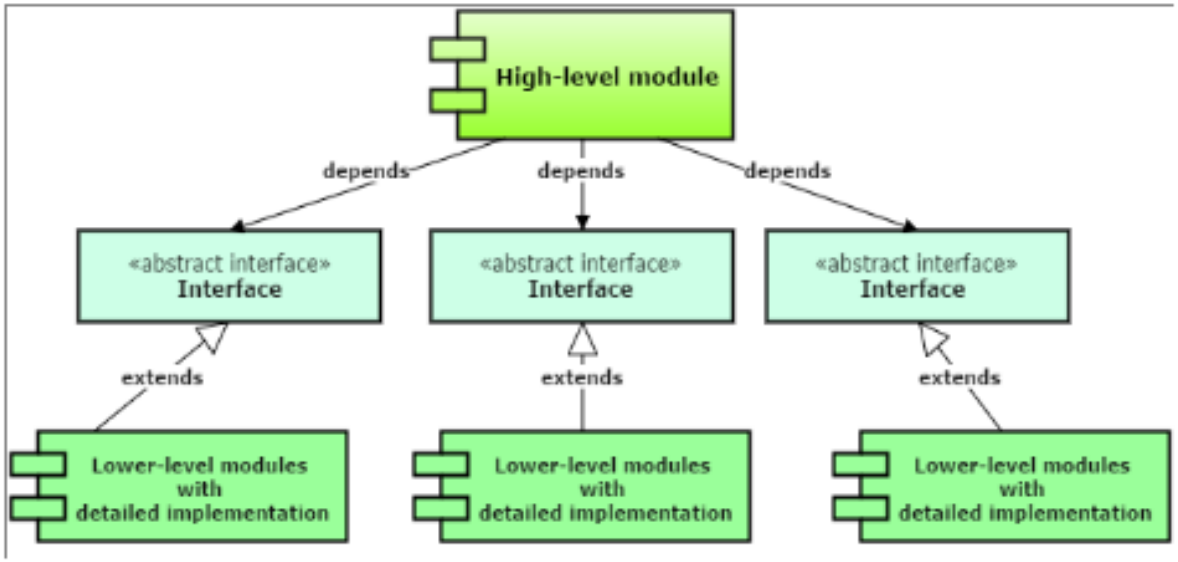
Dependency Inversion Principle

To understand the Dependency Inversion Principle better, let us first take a look at how the traditional procedural systems have their dependencies organised. In procedural systems, higher level modules depend on lower level modules to fulfil their responsibilities. The diagram below shows the procedural dependency structure



Dependency Inversion Principle

The Dependency Inversion Principle, however, advocates that the dependency structure should rather be inverted when designing object-oriented systems.. Have a look at the diagram below showing dependency structure for object-oriented systems



What is wrong with traditional procedural systems?

- If you take a re-look at the diagram above showing modular dependencies in a procedural system, then one can clearly see the tight coupling that each module layer has with its sub-layer.
- Thus, any change in the sub-layer will have a ripple effect in the next higher layer and may propagate even further upwards.
- This tight coupling makes it extremely difficult and costly to maintain and extend the functionality of the layers.

Advantage of using Dependency Inversion Principle

- The Dependency Inversion Principle, does away with this tight-coupling between layers by introducing a layer of abstraction between them.
- So, the higher-level layers, rather than depending directly on the lower-level layers, instead depend on a common abstraction.
- The lower-level layer can then vary(be modified or extended) without the fear of disturbing higher-level layers depending on it, as long as it obeys the contract of the abstract interface. If, as shown in the object-oriented design diagram above, the lower layers literally extend the abstraction layer interfaces, then they will follow the contract.

Solving Dependency Inversion

There may arise scenarios where the lower-level layers cannot directly extend the abstraction interfaces. In such cases an Adapter Design Pattern implementation is used.