Department Of Computer Science

Software Engineering – I (CSC291)

Lecture 06

Design

Objectives

- **Design:**
- **"Symptoms" of bad design**
  - Rigidity
  - Fragility
  - Immobility
  - Viscosity
- **Design Principle: SOLID**
  - **SRP** Single Responsibility Principle
  - **OCP** Open Closed Principle
  - **LSP** Liskov Substitution Principle
  - **ISP** Interface Segregation Principle
  - **DIP** Dependency Inversion Principle

Objectives

- **Design pattern**
  - **Motivation**
  - Properties of pattern
  - What is Gang of Four (GOF)?
  - **Different Categories of design patterns**
    - Creational Patterns
    - Structural Patterns
    - Behavioural Patterns
  - **Factory and Singleton pattern**
  - **Benefits of using design patterns**
  - **Elements of Design Patterns**
    - Pattern name: increases vocabulary of designers
    - Problem: intent, context, when to apply
    - Solution: UML-like structure, abstract code
    - Consequences: results and tradeoffs
  - **Examples**

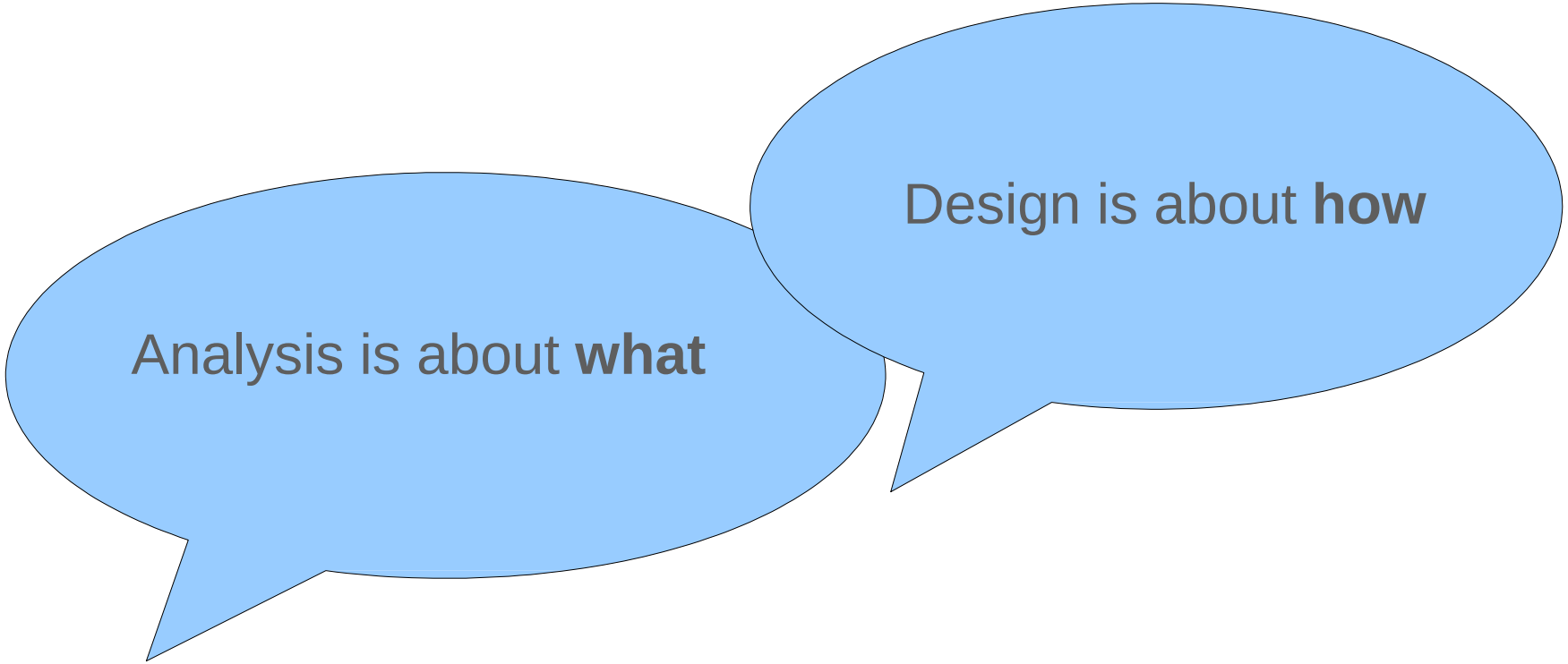
What is a Design?

- **Software design and implementation** is the stage in the software engineering process at which an executable software system is developed.
- **Software design and implementation** activities are invariably inter-leaved.
  - **Software design is a creative activity** in which you identify software components and their relationships, based on a customer's requirements.

  Implementation is the process of realizing the design as a program.

Design

- What's the meaning of design?
- What's the difference if compared to analysis?



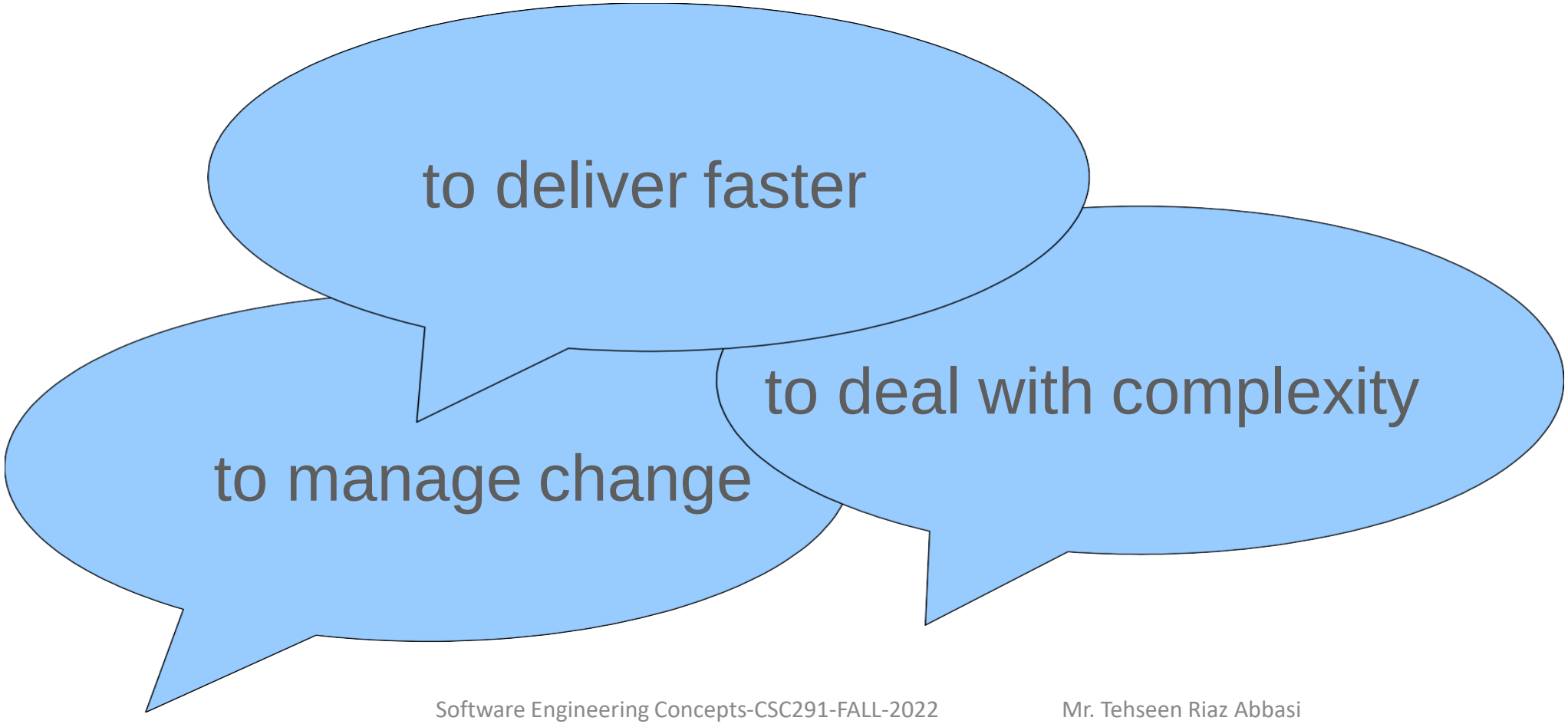
Analysis Vs Design

- **Analysis** is study of an entity.
- **Analysis** is examining and getting a better picture of the current system and its operations
- **Analysis** concentrates on refining the problems to solve
- **In UML**, analysis would be in the requirements, use case, activity diagram arena.

- **Design is** creation of an entity.
- **Design is** the actual developing of a blueprint of the new proposed system
- **Design focuses** on the solutions
- **Design would** be in the class diagram, sequence diagram, state diagram section.

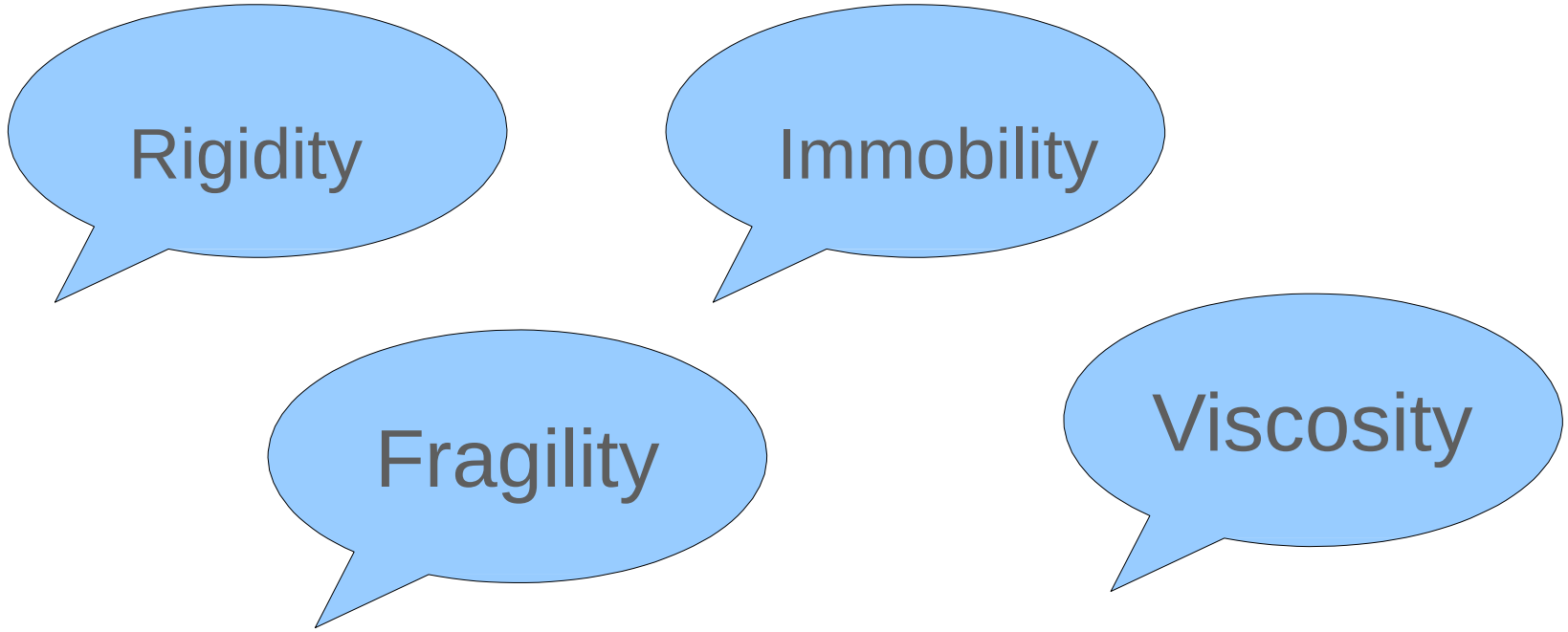
Design

- Why do we need (good) design?



Design

- Ok, we probably need better criteria
- **Are there any "symptoms" of bad design**



Rigidity

- **The impact of a change is unpredictable**
- It is hard to change because every change affects too many other parts of the system.
- Every change causes a cascade of changes in dependent modules
- A nice "two days" work become a kind of endless marathon
- Costs become unpredictable

Fragility

- **The software tends to break in many places on every change**
- When you make a change, unexpected parts of the system break.
- The breakage occurs in areas with no conceptual relationship
- On every fix the software breaks in unexpected ways

Immobility

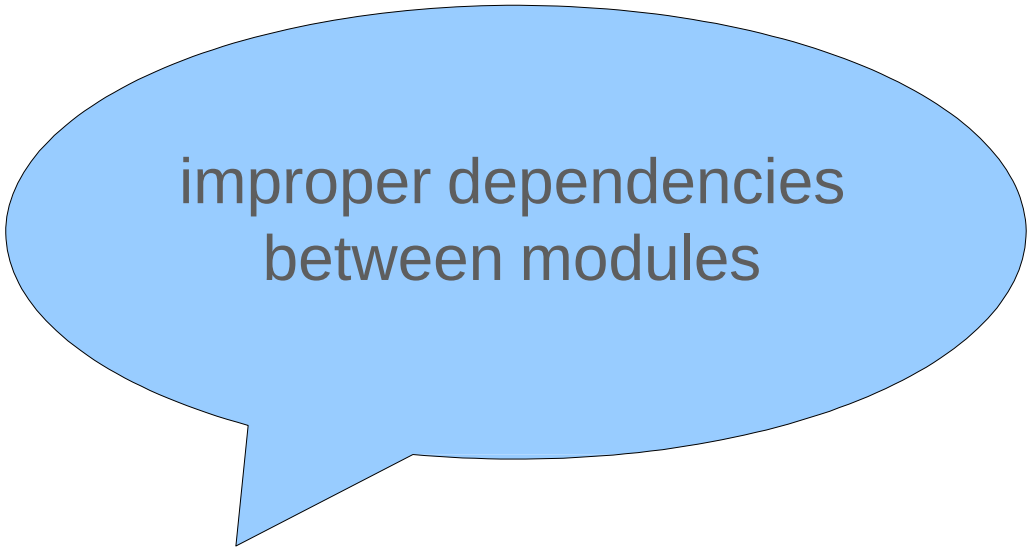
- **It's almost impossible to reuse interesting parts of the software**
- The useful modules have too many dependencies
- The cost of rewriting is less compared to the risk faced to separate those parts

Viscosity

- **A hack is cheaper to implement than the solution within the design**
- Preserving design moves are difficult to think and to implement
- It's much easier to do the wrong thing rather than the right one

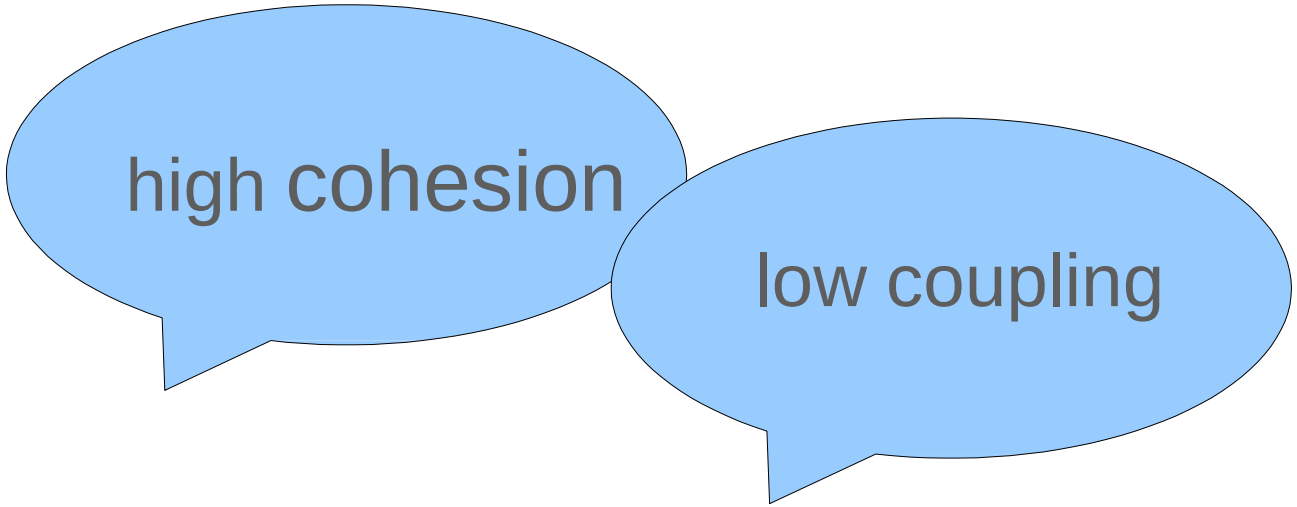
Design

- What's the reason why a design becomes rigid, fragile, immobile, and viscous?



Good design

- So, what are the characteristics of a good design?



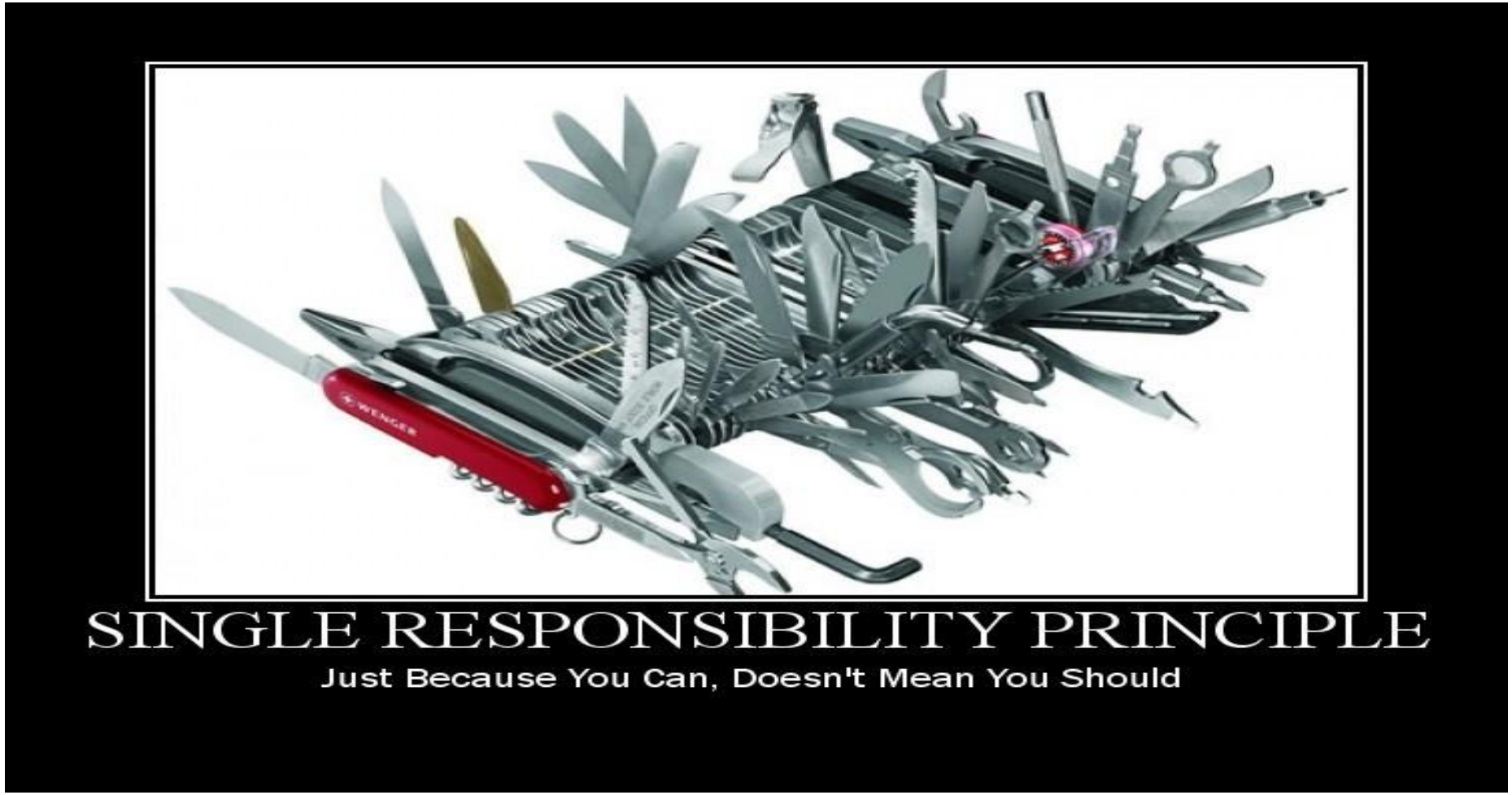
Good design

- How can we achieve a good design?

Let's go **SOLID**!!!

Design Principle : SOLID

- An acronym of acronyms!
- It recalls in a single word all the most important pricinple of design
  - **SRP** Single Responsibility Principle
  - **OCP** Open Closed Principle
  - **LSP** Liskov Substitution Principle
  - **ISP** Interface Segregation Principle
  - **DIP** Dependency Inversion Principle


SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

Single Responsability Principle

- **A class should have only one reason to change.**
- This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes.
- It translates directly in high cohesion
- In class having more responsibilities, the change might affect the other functionality of the classes.

SRP

- Identify things that are changing for different reasons
- Group together things that change for the same reason
- Introduced by Tom DeMarco in his book Structured Analysis and Systems Specification, 1979

SRP Example

```
class Employee {
   public Pay calculatePay() {...}
   public void save() {...}
   public String
describeEmployee() {...}
}
```

- How many responsibilities???


OPEN CLOSED PRINCIPLE
Open Chest Surgery Is Not Needed When Putting On A Coat

Open Closed Principle

- **Software entities like classes, modules and functions should be open for extension but closed for modifications.**
- Theorized in 1998 by Bertrand Meyer in a classical OO book
- You should be able to extend the behavior of a module without changing it!

OCP

- Abstraction is the key!
- Keep the things that change frequently away from things that don't change
- If they depend on each other, things that change frequently should depend upon things don't change

OCP Example

```
void checkOut(Receipt receipt) {
    Money total = Money.zero;
    for (item : items) {
        total += item.getPrice();
        receipt.addItem(item);
    }
    Payment p = acceptCash(total); receipt.addPayment(p);
}
```

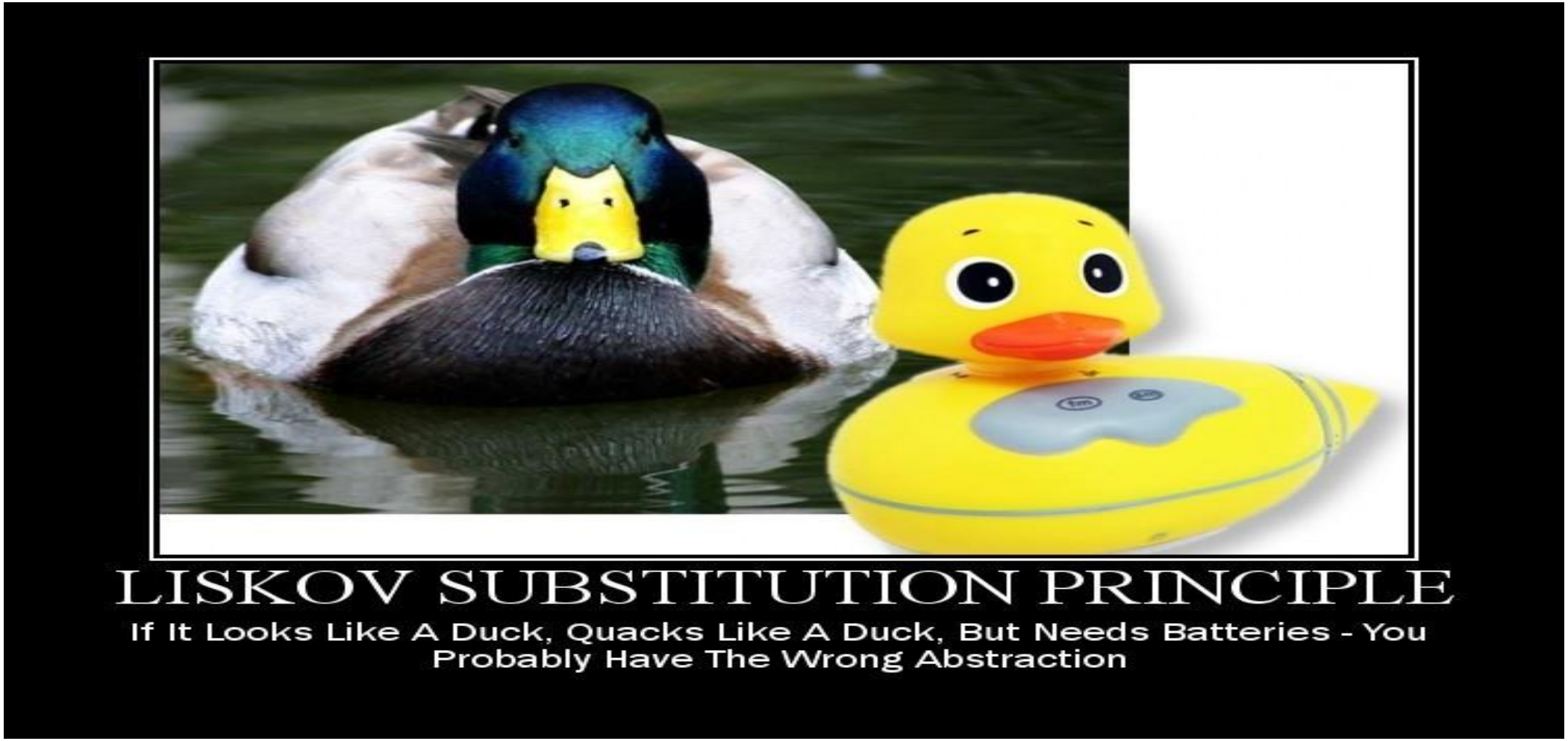
What if we want to add acceptCreditCard method???

OCP Example

```
public interface PaymentMethod {
        void acceptPayment(Money total);
}
void checkOut(Receipt receipt, PaymentMethod pm) {
        Money total = Money.zero;
        for (item : items) {
                total += item.getPrice();
        receipt.addItem(item);
        }
        Payment p = pm.acceptPayment(total);
receipt.addPayment(p);
}
```

OCP Example

```
public class Cash implements PaymentMethod {
        void acceptPayment(Money total){
        /// implementation
}}
public class CreditCard implements PaymentMethod {
        void acceptPayment(Money total){
        /// implementation
}}
Public static void main (String arg[]){
        PaymentMethod  c = new Cash();
        PaymentMethod  credit  = new CreditCard();
    checkOut(receipt, c);
    //checkOut(receipt, credit);
}
```


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Liskov Substitution Principle

- Derived types must be completely substitutable for their base types.
- This principle is just an extension of the Open Close Principle in terms of behavior.
- It means that we must make sure that new derived classes are extending the base classes without changing their behavior.
- The new derived classes should be able to replace the base classes without any change in the code.

LSP (by example)

- **How would** you model the relationship between a square and a rectangle?
- **Should** the square class extends rectangle?
- **Of course**, isn't the Square a kind of Rectangle, after all?
- **It seems an obvious IS A**
- **But... what about:**
  - rectangle has two attributes, width and height: how can we deal with that?
  - **how do we deal** with setWidth() and setHeight() ?
- Is it safe?

LSP (by example)

- No, behavior is different
- If I pass a Square to a Rectangle aware function, then this may fail as it may assume that width and heigth are managed separately

LSP (by example)

```
class Rectangle {
    protected int m_width;
    protected int m_height;
    public void setWidth(int width){
        m_width = width;
    }
    public void setHeight(int height){
        m_height = height; }
    public int getWidth(){
        return m_width; }
    public int getHeight(){
        return m_height; }
    public int getArea(){
        return m_width * m_height; }
}
```

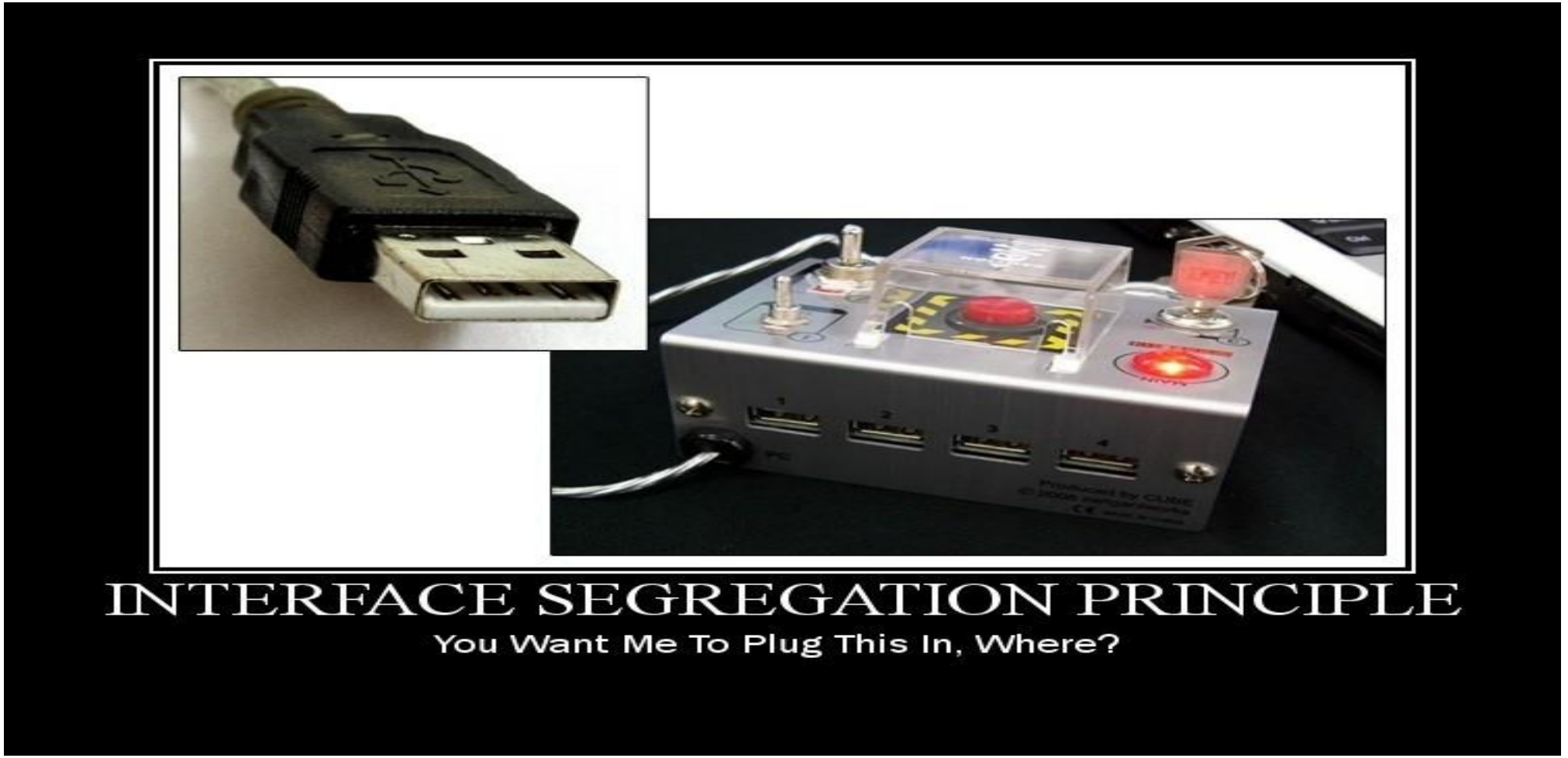
LSP (by example)

```
class Square extends Rectangle {
    public void setWidth(int width){
        m_width = width;
        m_height = width;
    }
    public void setHeight(int height){
        m_width = height;
        m_height = height; }
}
```

LSP (by example)

```
class LspTest {
    private static Rectangle getNewRectangle() {
            return new Square();
    }

    public static void main (String args[]) {
    Rectangle r = LspTest.getNewRectangle();
    r.setWidth(5);
    r.setHeight(10);
    // user knows that r it's a rectangle.
    // It assumes that he's able to set the width and height as for the // base class
    System.out.println(r.getArea());
    // now he's surprised to see that the area is 100 instead of 50. }
}
```

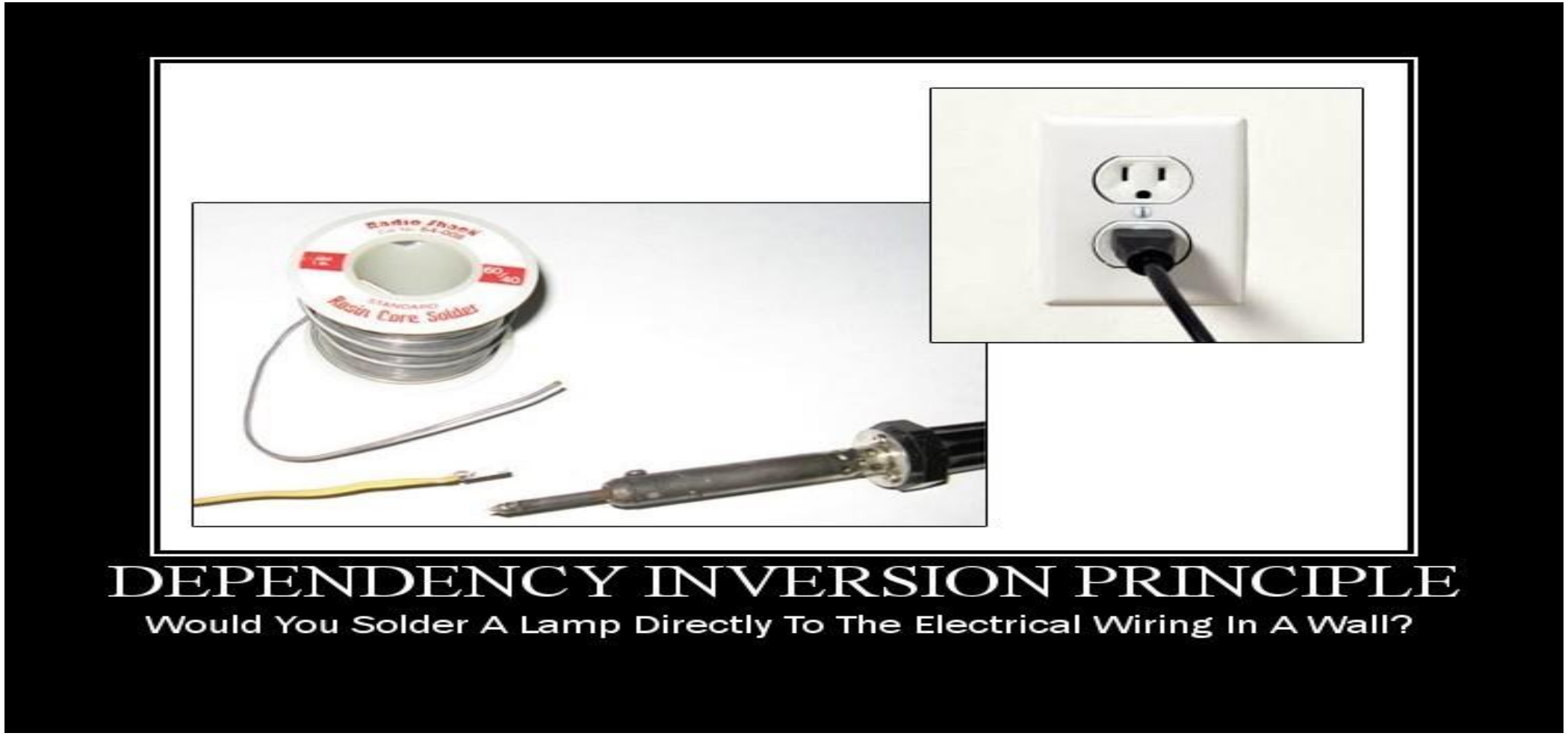

INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

Interface Segregation Principle

- Clients should not be forced to depend upon interfaces that they don't use.
- Interfaces should only contain methods that should be there.
- Otherwise, classes implementing the interface will have to implement those methods as well.
- For example if we create an interface called Worker and add a method lunch break, all the workers will have to implement it. What if the worker is a robot?

ISP

- Interfaces containing methods that are not specific to it are called polluted or fat interfaces.
- ISP states that clients should not know about fat classes
- Instead they should rely on clean cohesive interfaces
- You don't want to depend upon something you don't use


Radio Shack
60/40
STANDARD
Rosin Core Solder
DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle

- High level modules should not depend upon low level modules, both should depend upon abstractions
- Abstractions should not depend upon details, details should depend upon abstractions

DIP

- **Don't depend on anything concrete, depend only upon abstraction**
- High level modules should not be forced to change because of a change in low level / technology layers
- Drives you towards low coupling

DIP

```
// Dependency Inversion Principle - Bad example
    class Worker {
            public void work() {
            // ....working
            }
    }
    class Manager {
            Worker worker;
            public void setWorker(Worker w) {
            worker = w;
            }
            public void manage() {
            worker.work();
            }
    }
    class SuperWorker {
            public void work() {
            //.... working much more
            }
    }
```

Low Level Class

High Level Class

Low Level Class

DIP

```
// Dependency Inversion Principle - Good example
    interface IWorker {
            public void work();
    }
    class Worker implements IWorker{
            public void work() {
            // ....working
            }
    }
    class SuperWorker implements IWorker{
            public void work() {
                    //.... working much more
            }
    }
    class Manager {
            IWorker worker;
            public void setWorker(IWorker w) {
                    worker = w;
            }
            public void manage() {
                    worker.work();}
    }
```

Low Level Class

Low Level Class

High Level Class

Design Principles Summary

- Single Responsibility Principle
  - A class should have only one reason to change.
- Open Close Principle
  - Software entities like classes, modules and functions should be open for extension but closed for modifications.
- Liskov's Substitution Principle
  - Derived types must be completely substitutable for their base types.
- Interface Segregation Principle
  - Clients should not be forced to depend upon interfaces that they don't use.
- Dependency Inversion Principle
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend on details. Details should depend on abstractions.


Thank
you