SOLID: Principles of OOD

CS480 Software Engineering
http://cs480.yusun.io
February 18, 2015

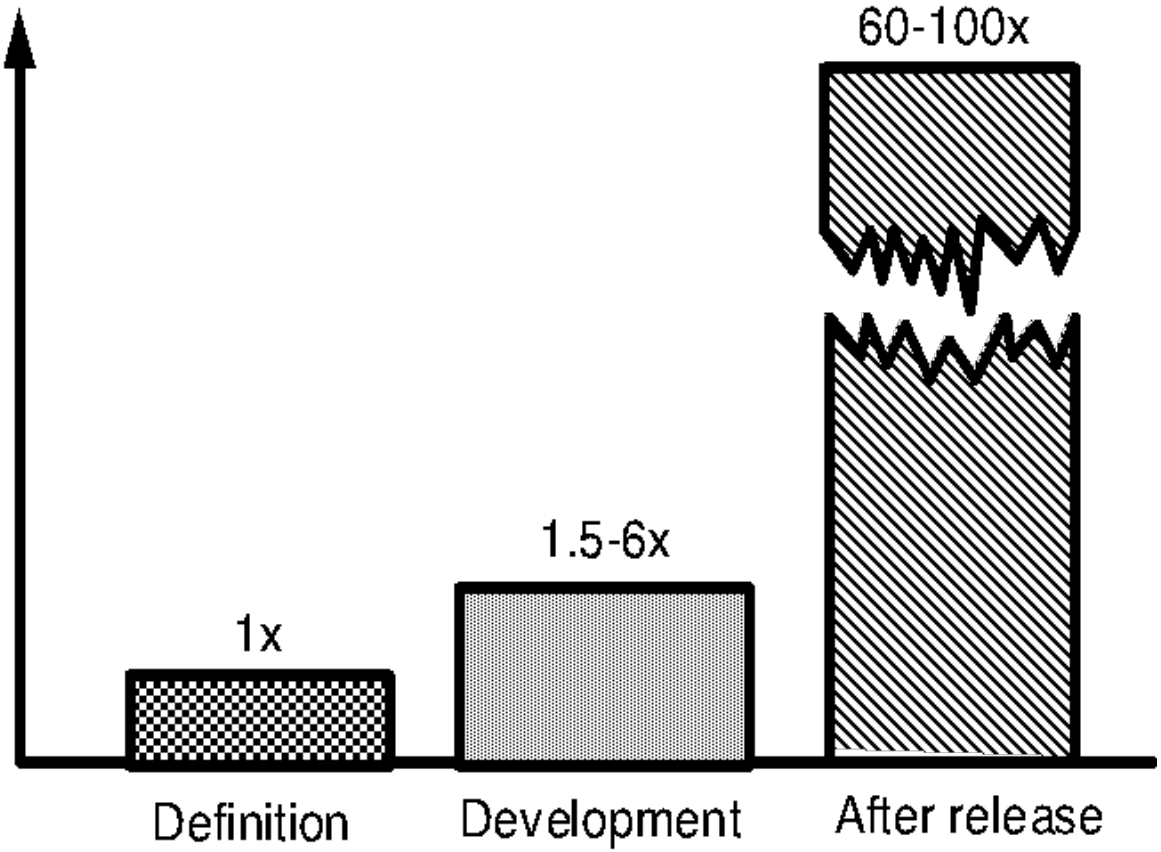
Yu Sun, Ph.D.
http://yusun.io
yusun@cpp.edu

CAL POLY POMONA

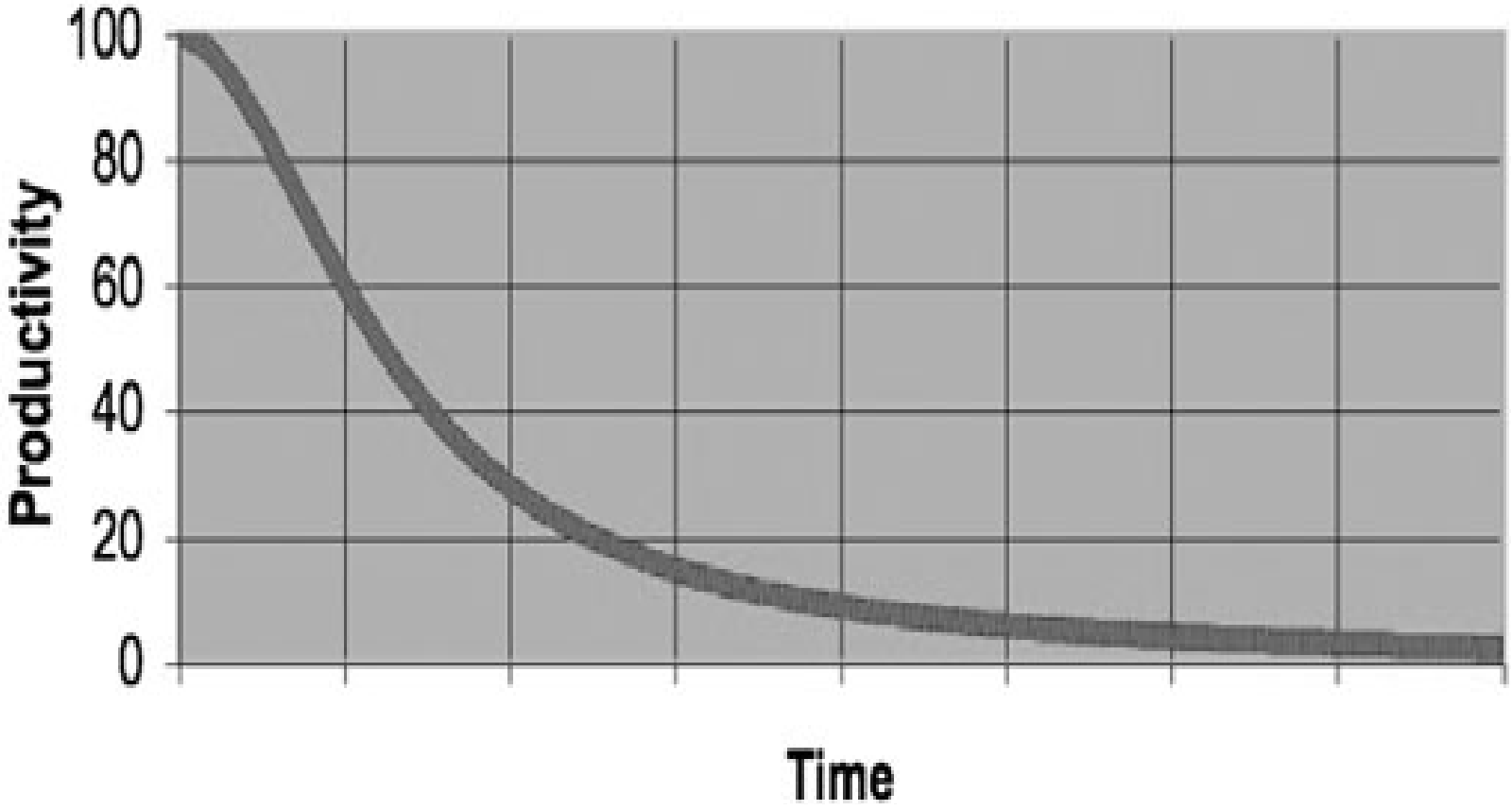
Software Nature – Software Entropy

- Software tends to degrade / decay
- Software rot – like a piece of bad meat

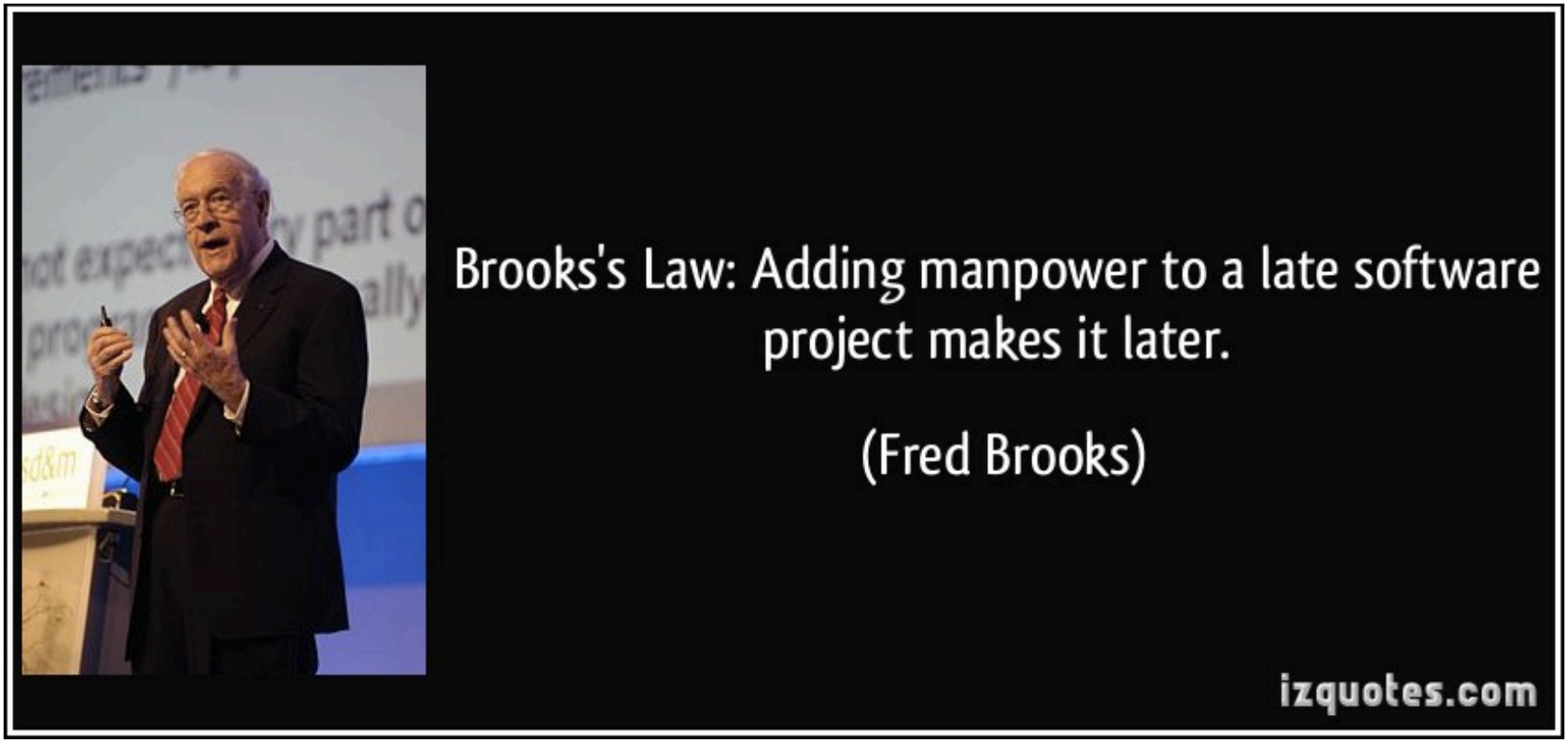
The Cost of Change



Developers Productivity vs. Time



Add More Developers ?

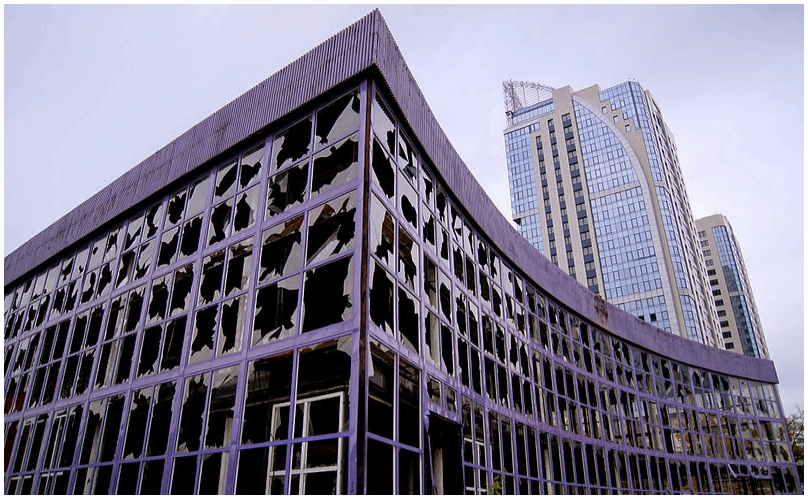


What Stimulates the Software to Rot?

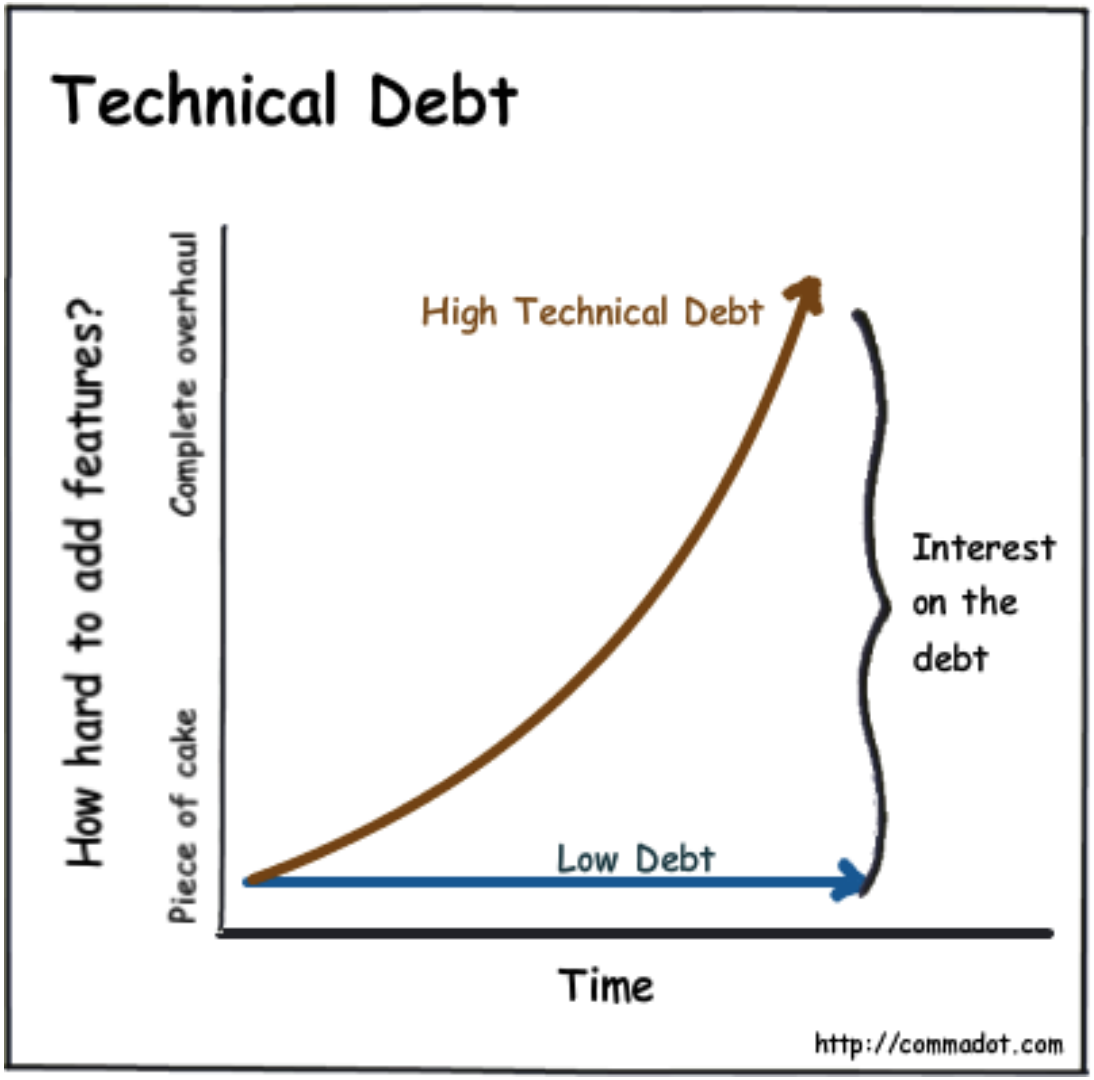
- Poor developers
- Requirements keep change – design degradation
- People change – violate the original design
- Tight schedule pressure

Psychology Reason: Broken Window Theory

- Came from city crime researcher
- A broken window will trigger a building into a smashed and abandoned derelict
- So does the software
- Don't live with the Broken window



Tech Debt


Technical Debt
How hard to add features?
Complete overhaul
Piece of cake
High Technical Debt
Low Debt
Interest on the debt
Time
http://commadot.com


DEBT

How to Prevent Software from Rotting?

- Applies OO design principles
- Uses design patterns
- Follows agile practices
- Refactoring will reduce the software entropy

S.O.L.I.D Design Principles

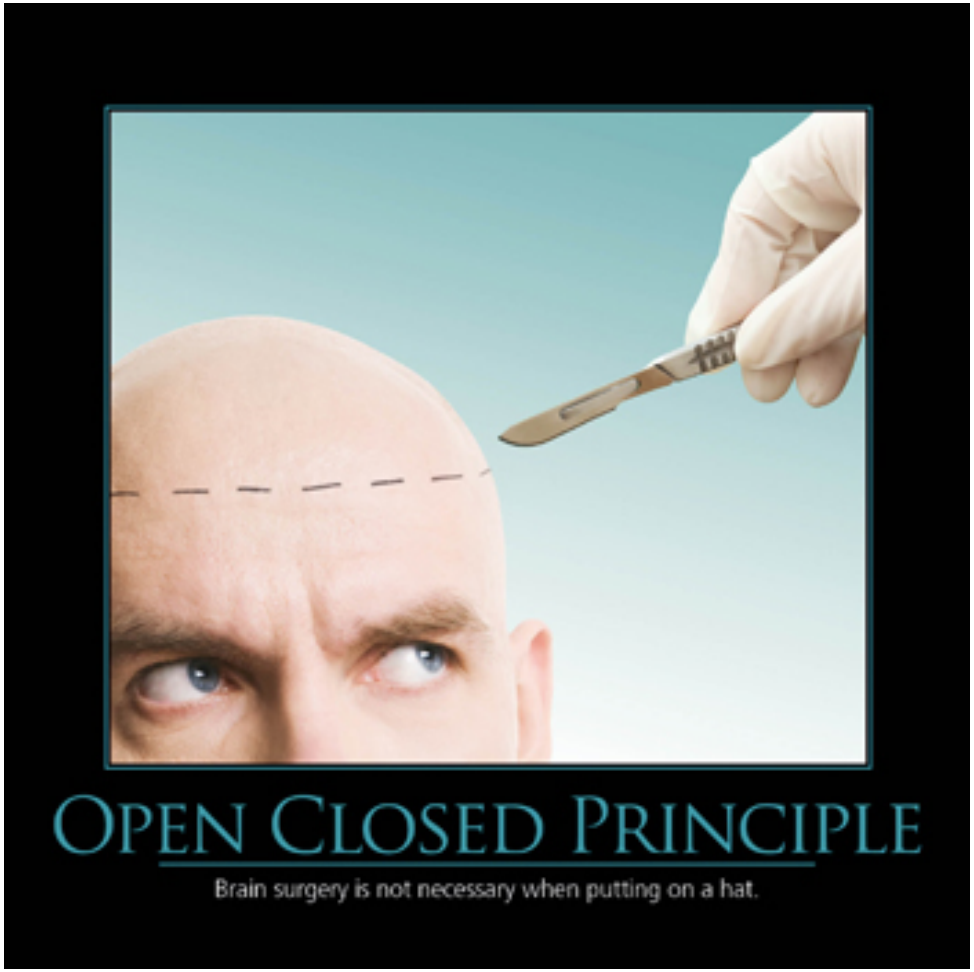
SOLID

Software Development is not a Jenga game

S.O.L.I.D Design Principles

- SRP – The **S**ingle Responsibility Principle
- OCP – The **O**pen-Closed Principle
- LSP – The **L**iskov Substitution Principle
- ISP – The **I**nterface Segregation Principle
- DIP – The **D**ependency Inversion Principle

1. Open Close Principle



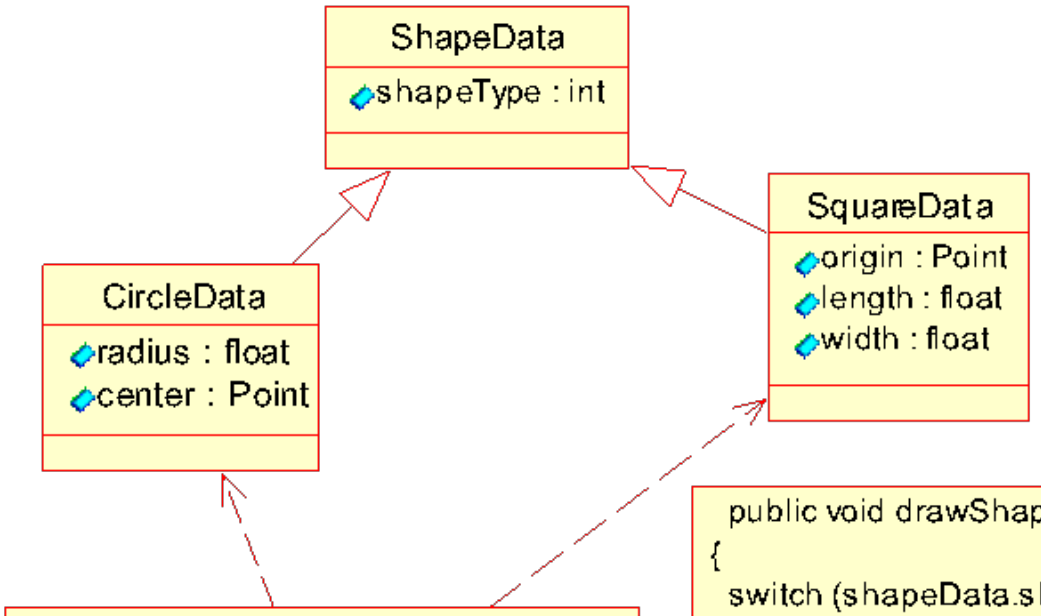
Open-Closed Principle (OCP)

> *Software entities should be open for extension, but closed for modification*
>
> B. Meyer, 1988 / quoted by R. Martin, 1996

- "*Software Systems change during their life time*"
  - Both better designs and poor designs have to face the changes;
  - Good designs are stable

- Be open for extension
  - Module's behavior can be extended
- Be closed for modification
  - Source code for the module must not be changed

The Open/Closed Principle (OCP) Example

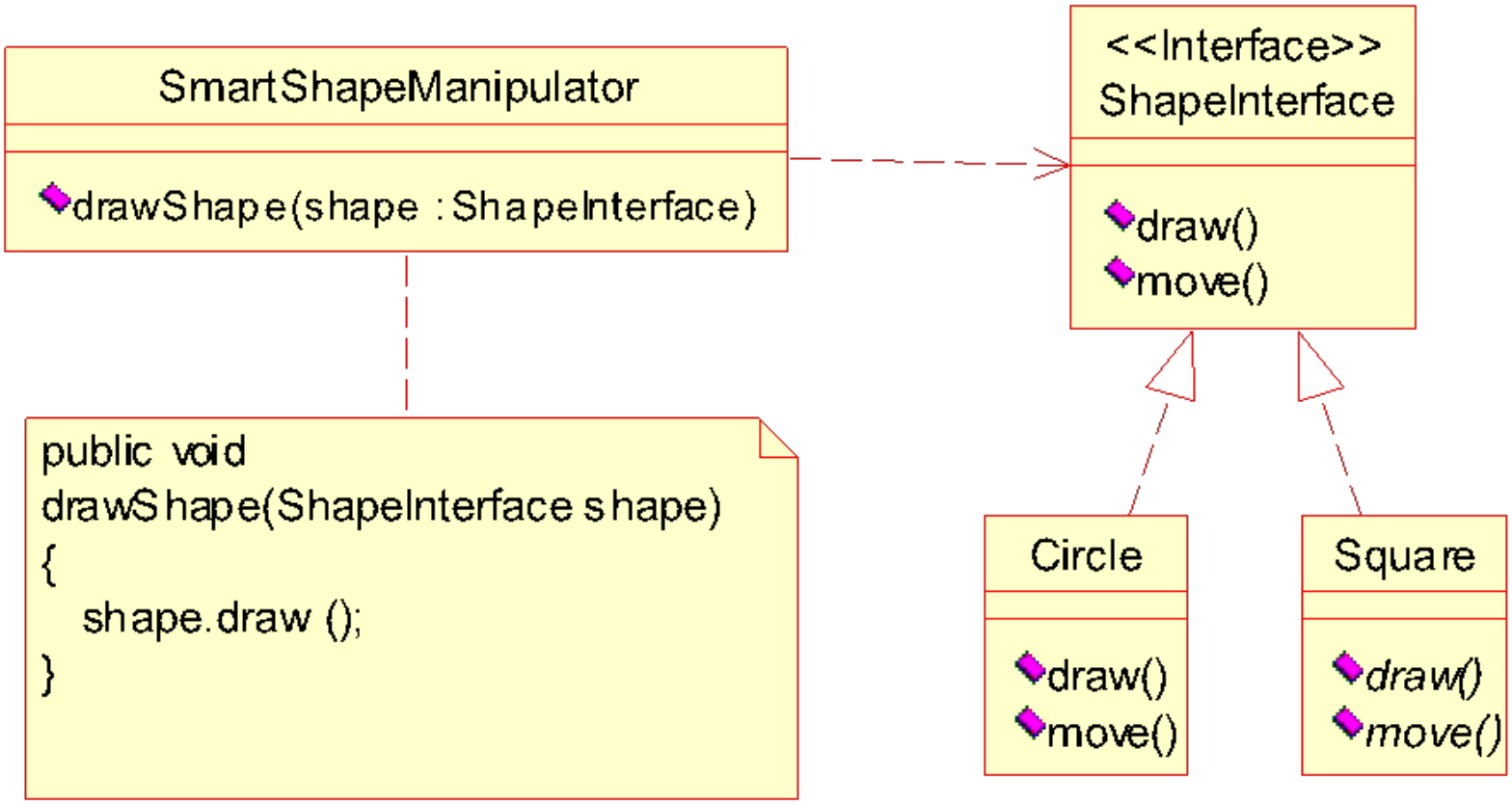


An Example of what not to do!

What is wrong with this code?

```
public void drawShape(ShapeData shapeData)
{
 switch (shapeData.shapeType)
 {
        case SQUARE:
                drawSquare((SquareData)shapeData);
                break;
        case CIRCLE:
                drawCircle((CircleData)shapeData);
                break;
 }
}
```

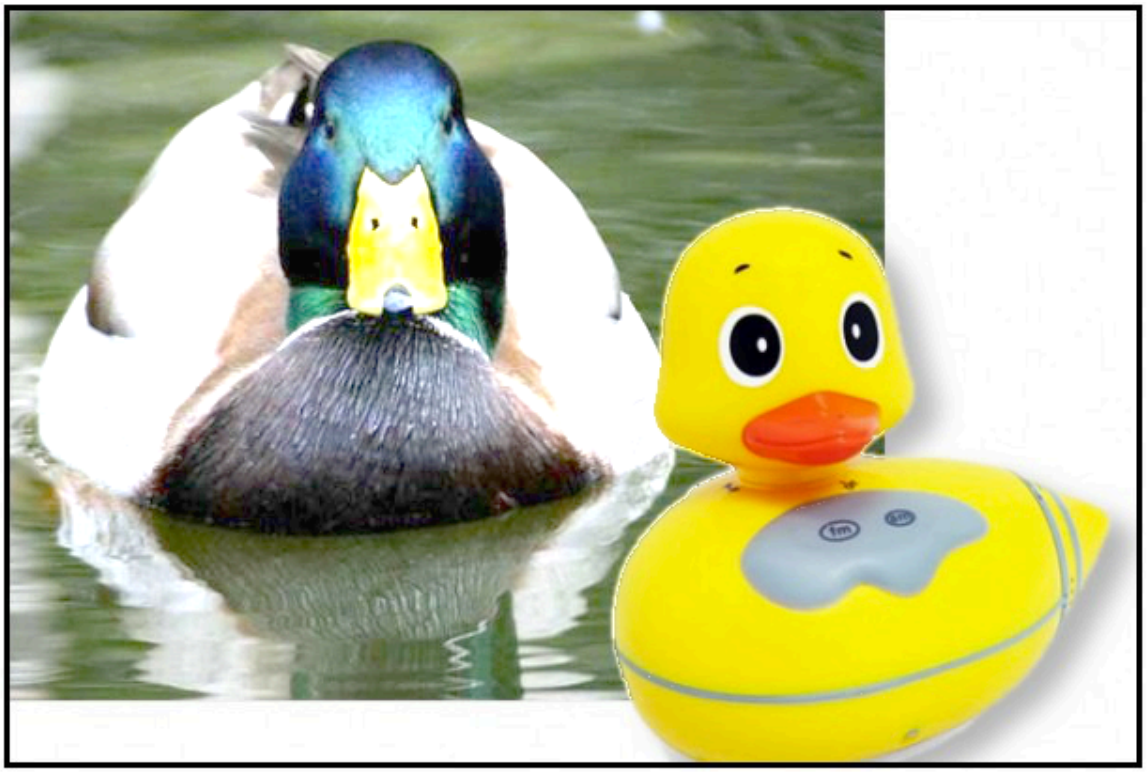
The Open/Closed Principle (OCP) Example



OCP Example

- Different CD/Radio/MP3 players can be plugin to the car dashboard.
- …using polymorphic dependencies

2. Liskov Substitution Principle



LISKOV SUBSTITUTION PRINCIPLE

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

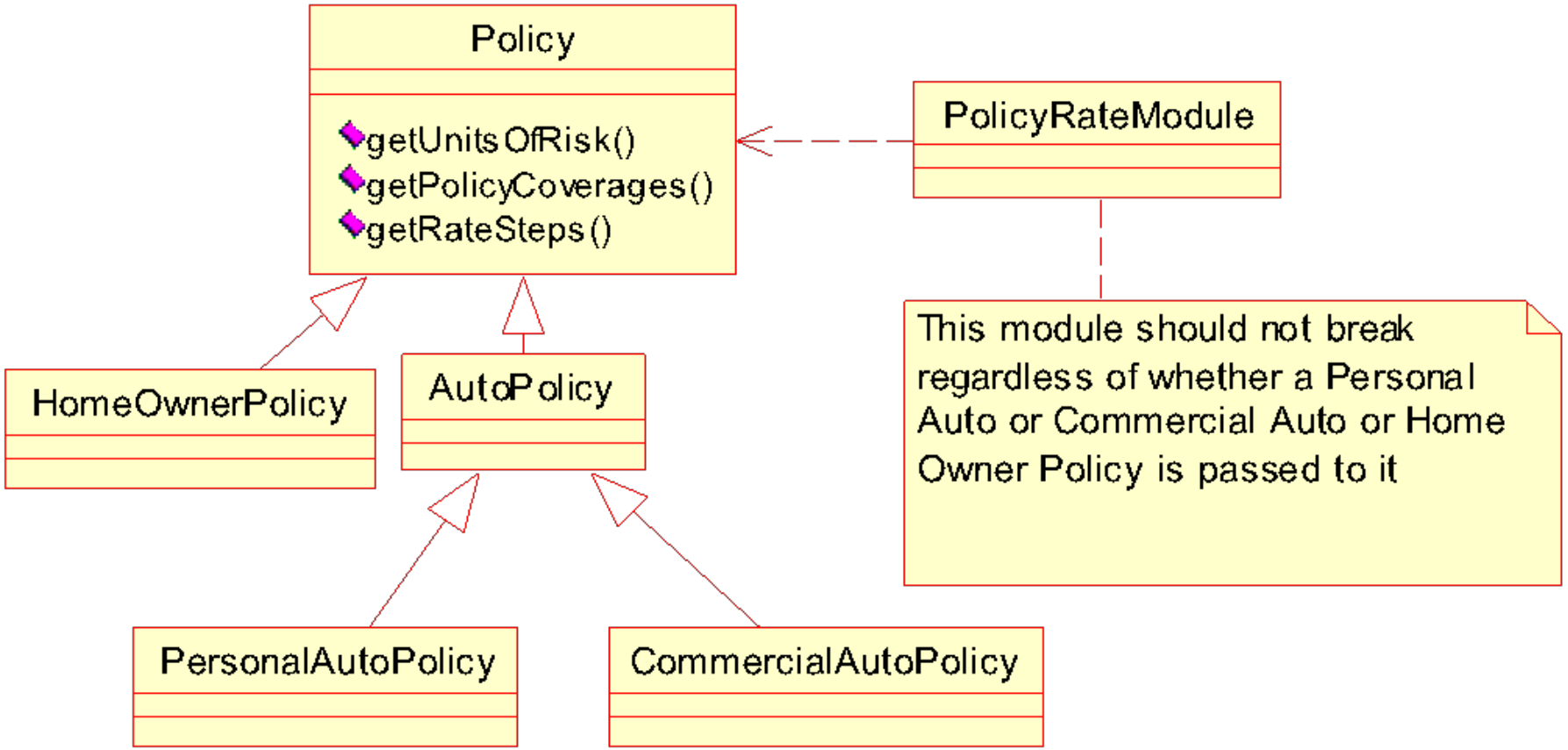
Liskov Substitution Principle (LSP)

- The key of OCP: Abstraction and Polymorphism
  - Implemented by inheritance
  - How do we measure the quality of inheritance?

> *Inheritance should ensure that any property proved about supertype objects also holds for subtype objects*
>
> **B. Liskov**, 1987

> *Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it*
>
> **R. Martin**, 1996

The Liskov Substitution Principle (LCP) Example



Inheritance *Appears* Simple

```
interface Bird {                    // has beak, wings,...
  public void fly();                // Bird can fly
}

class Parrot implements Bird {    // Parrot is a bird
  public void fly() { … }       // Parrot can fly
  public void mimic() { … };  // Can Repeat words...
}

// ...
Parrot mypet;
mypet.mimic();    // my pet being a parrot can Mimic()
mypet.fly();      // my pet “is-a” bird, can fly
```

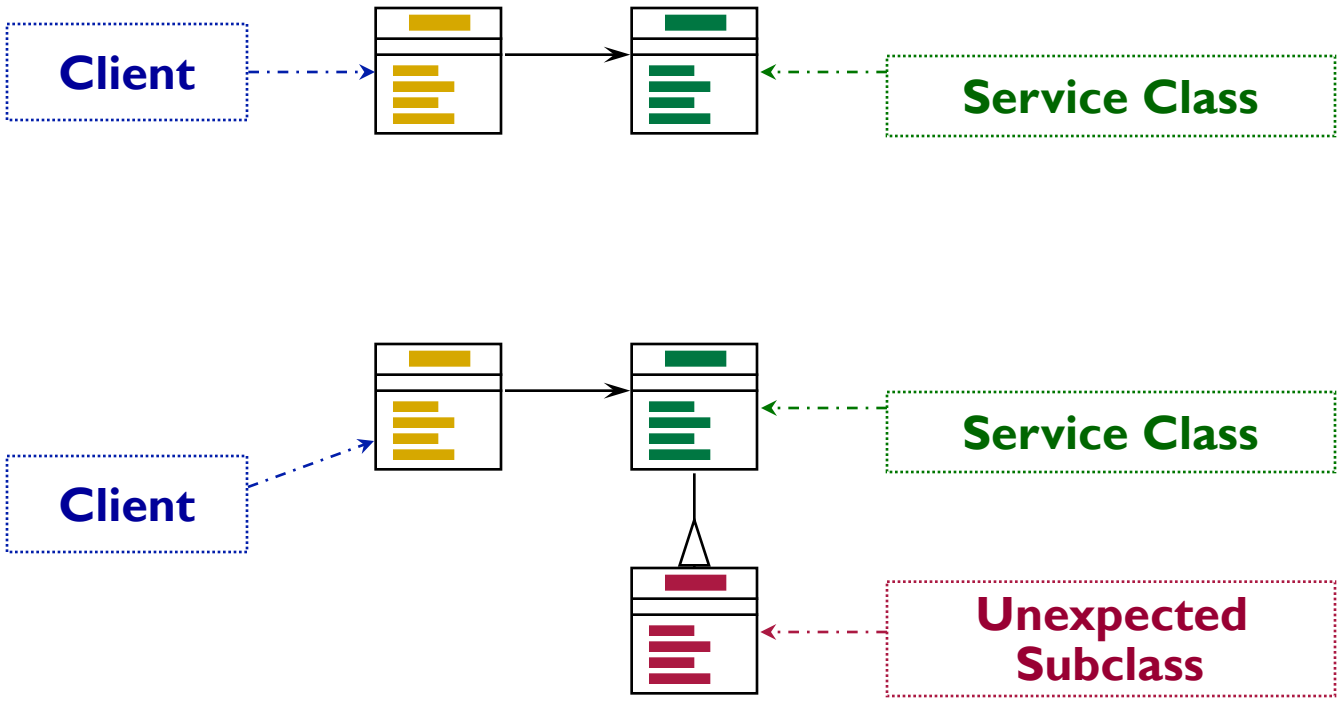
Penguins Fail to Fly!

```
class Penguin implements Bird {
   public void fly() {
     error ("Penguins don't fly!"); }
}

void PlayWithBird (Bird abird) {
    abird.fly();    // OK if Parrot.
    // if bird happens to be Penguin...OOOPS!!
}
```

- Does not model: "Penguins can't fly"
- It models "Penguins may fly, but if they try it is an error"
- Run-time error if attempt to fly → not desirable
- Think about Substitutability – Fails LSP

LSP and Replaceability

- Any code which can legally call another class's methods
  - Must be able to substitute any subclass of that class without modification:



3. Single Responsibility Principle

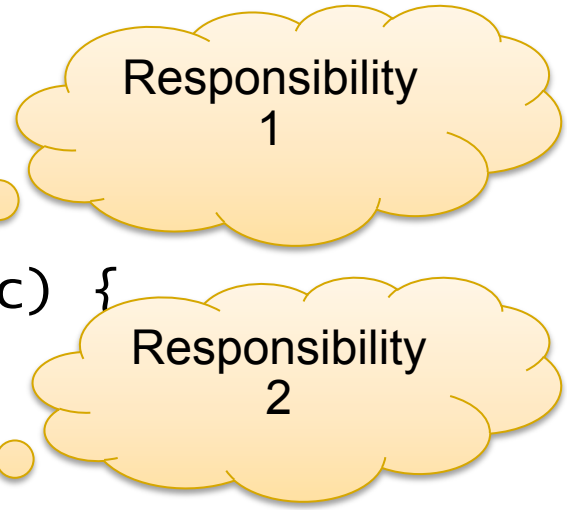
Can’t you do anything right?

What's the Issue?

```
Public class Customer {
    private String name;
    private String address;

    public void addCustomer(Customer c) {
        // database code goes here
    }

    public void generateReport(Customer c) {
        // set report formatting
    }
}
```

What does the following code do?

```
Public class Customer {
    private String name;
    private String address;

    public void addCustomer(Customer c) {
        // database code goes here
    }

    public void generateReport(Customer c) {
        // set report formatting
    }
}
```



Every time one gets changed there is a chance that the other also gets changed because both are staying in the same home and both have same parent. We can't control everything. So a single change leads to double testing (or maybe more).

OVERLOAD Kills

What is SRP?

> Every software module should have only one reason to change
>
> **R. Martin**

- Software Module – Class, Function, etc.
- Reason to Change – Responsibility

Solution which will not violate SRP

```
Public class Customer {
    private String name;
    private String address;
    // setter and getter methods
}

public class CustomerDB {
    public void addCustomer(Customer c) {
        // database login goes here
    }
}

public class CustomerReport {
    public void generateReport(Customer c) {
        // set report formatting
    }
}
```

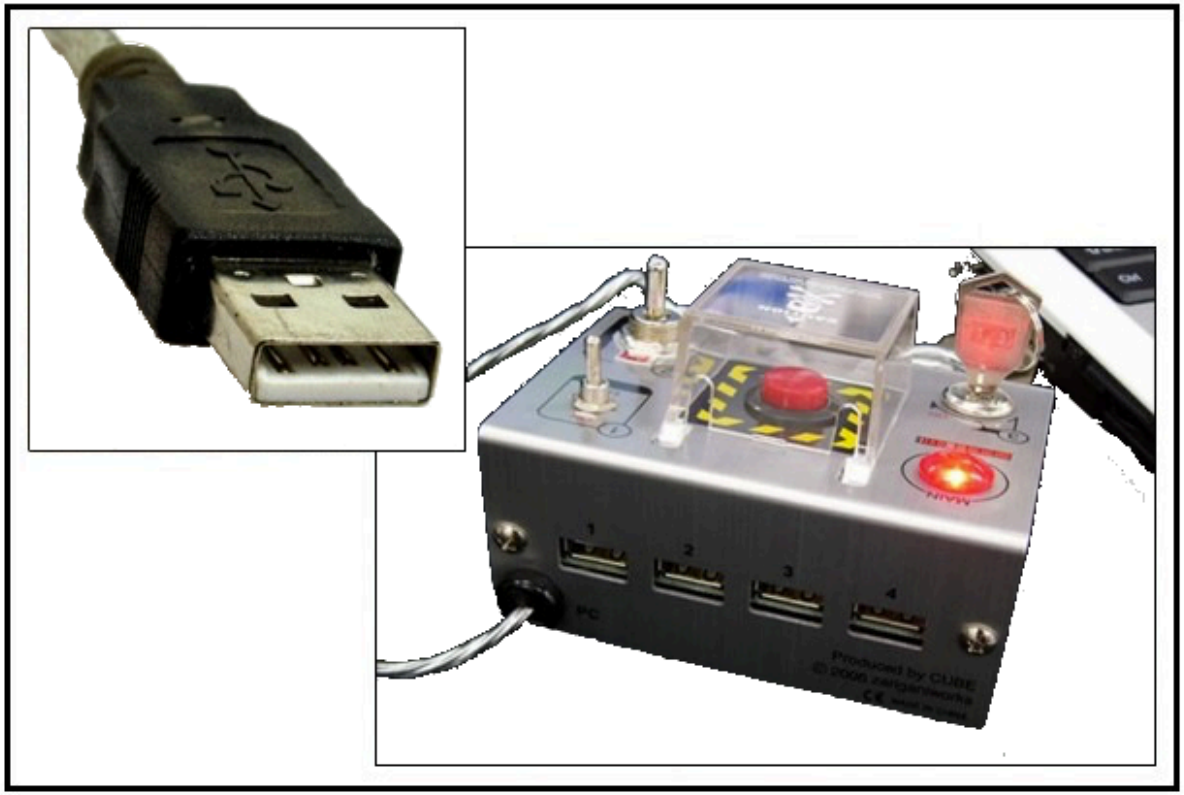
Can a single class has multiple methods?

- YES!
- The class responsibility is described at a higher level, or is related to the context

```
public class CustomerDB {
    public void addCustomer(Customer c) {
        // database logic goes here
    }
    public Customer getCustomer(String name) {
         // database logic goes here
    }
}

public class CustomerReport {
    public void generateReport(Customer c) {
        // set report formatting
    }
    public void persistReport(Custerom c) {
        // save report in disk
    }
}
```

Rule: Keep It Simple Stupid

4. Interface Segregation Principle



Real World Comparison

Report Management System

```
public class EmployeeUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
public class ManagerUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport ();
        objBal.GenerateProjectSchedule ();
    }
}
public class AdminUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
}
```

```
public interface IReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();

    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();

    void GenerateProfitReport();
}
```

IReportBAL is used by all the 3 components:

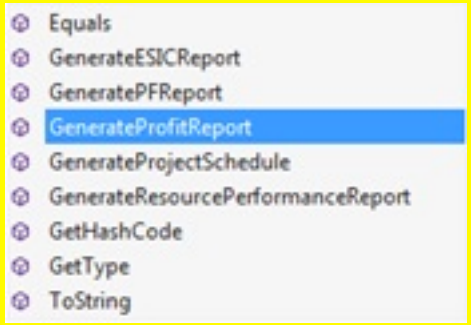
1. EmployeeUI
2. ManagerUI
3. AdminUI

The Problem

```
public class EmployeeUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
public class ManagerUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceRep
        objBal.GenerateProjectSchedule ();
    }
}
public class AdminUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
}
```

```
public interface IReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();

    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();

    void GenerateProfitReport();
}
```

Everytime "objBal" is typed, all the methods will be shown, which is not always necessary:



What is ISP?

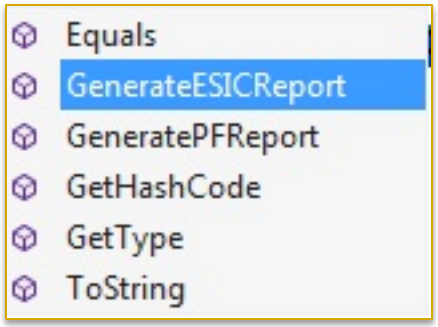
> Clients should not be forced to depend upon interfaces that they do not use.
>
> **R. Martin**

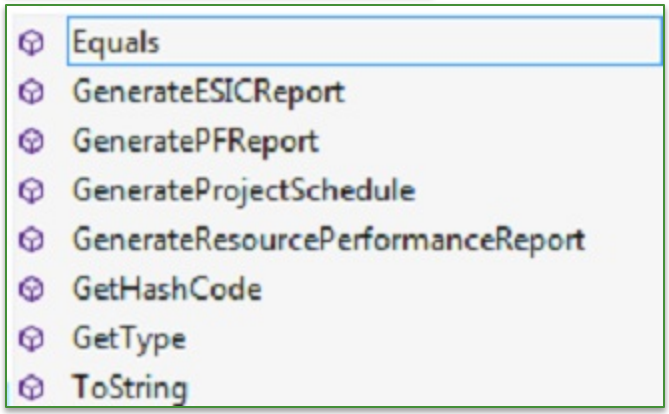
- Keep the interfaces concise and small

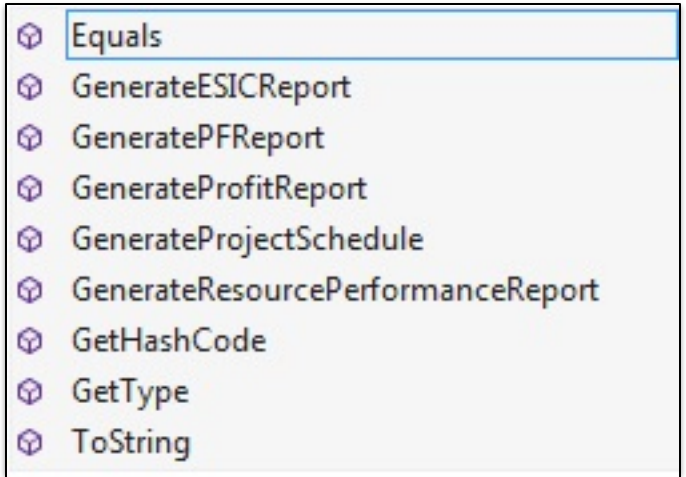
Interface Segregation

```
public interface IEmployeeReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();
}
public interface IManagerReportBAL : IEmployeeReportBAL
{
    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();
}
public interface IAdminReportBAL : IManagerReportBAL
{
    void GenerateProfitReport();
}
public class ReportBAL : IAdminReportBAL
{
    public void GeneratePFReport()
    {/*..............*/}

    public void GenerateESICReport()
    {/*..............*/}

    public void GenerateResourcePerformanceReport()
    {/*..............*/}

    public void GenerateProjectSchedule()
    {/*..............*/}

    public void GenerateProfitReport()
    {/*..............*/}
}
```

Interface Segregation





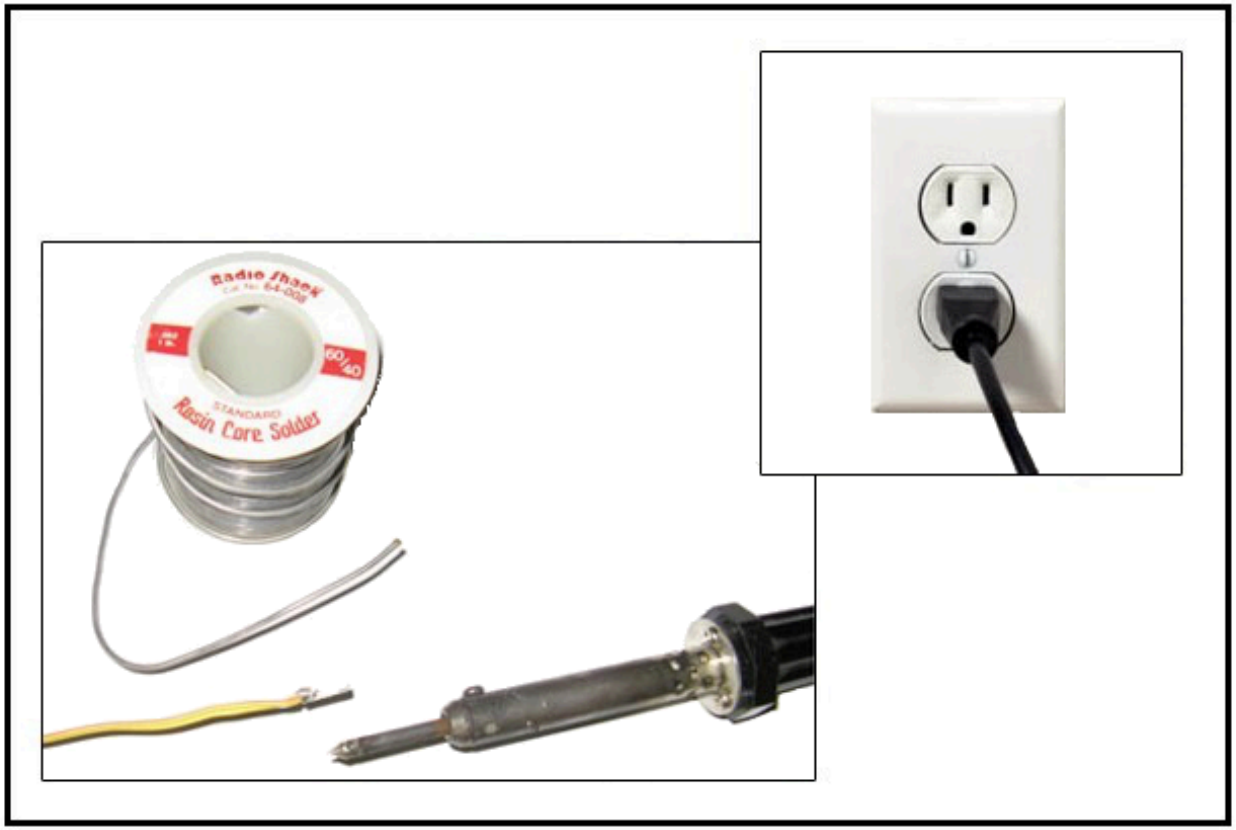



```
public class EmployeeUI
{
    public void DisplayUI()
    {
        IEmployeeReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
```

```
public class ManagerUI
{
    public void DisplayUI()
    {
        IManagerReportBAL  objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport ();
        objBal.GenerateProjectSchedule ();
    }
}
```

```
public class AdminUI
{
    public void DisplayUI()
    {
        IAdminReportBAL  objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
```

5. Dependency Inversion Principle



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle

I. High-level modules should ***not*** depend on low-level module implementations. Both levels should depend on abstractions

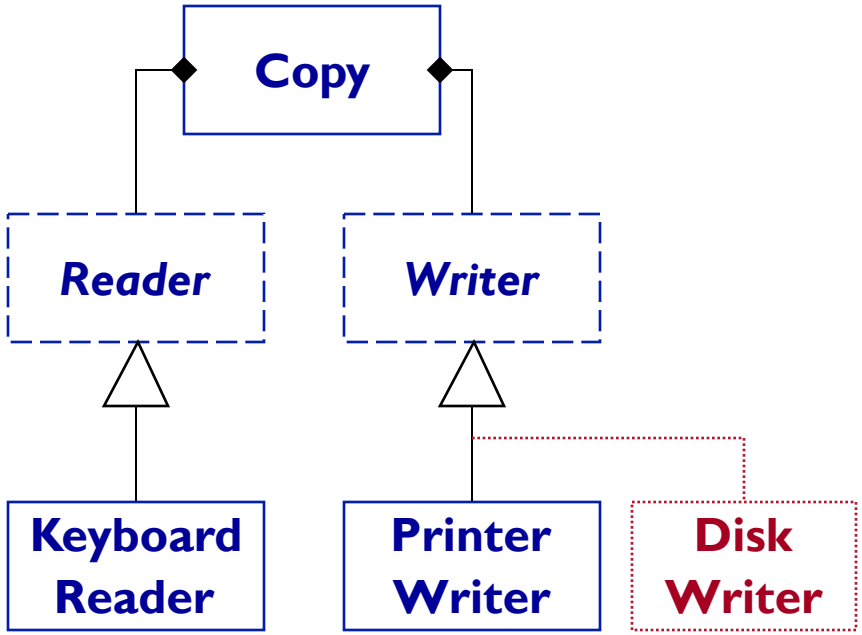
II. Abstractions should not depend on details

Details should depend on abstractions

R. Martin, 1996

- OCP states the goal; DIP states the mechanism
- A base class in an inheritance hierarchy should not know any of its subclasses
- Modules with detailed implementations are not depended upon, but depend themselves upon higher abstractions

DIP Applied on Example




```
class Reader {
  public:
    virtual int read()=0;
};

class Writer {
  public:
    virtual void write(int)=0;
};

void Copy(Reader& r, Writer& w){
   int c;
   while((c = r.read()) != EOF)
     w.write(c);
}
```

DIP Related Heuristic

Program to interface,
Not implementation!

- Use inheritance to avoid direct bindings to classes: