# Object Oriented Design Principles with C#

## Part – I

**Md. Mahedee Hasan**

**Software Architect**

**Leadsoft Bangladesh Limited**

**Microsoft MVP | Trainer | Technical Speaker**

Linkedin: http://www.linkedin.com/in/mahedee

Blog: http://mahedee.net/

http://mahedee.blogspot.com/

# Contents

- Introduction
- What is Object Oriented Design Principle?
- SOLID Object Oriented Design Principles
- SRP – Single Responsibility Principle
- OCP – Open Close Principle
- LSP – Liskov Substitution Principle
- ISP - Interface Segregation principle
- DIP – Dependency Inversion Principle

# Introduction

- **What is Principles?**
  - **Do** these and you will **achieve** this.
    - How you will do it is up to you.
  - Everyone defines some **principles** in their **lives** like
    - "I never lie“
    - “I never drink alcohol”

  - He/she **follow these principles** to make his/her **life easy**
  - How will he/she **stick** to these principles is up to the individual.

# What is Object Oriented Design Principle?

- **What is Object Oriented Design?**
  - It's **a process of planning a software system** where objects will interact with each other to solve **specific problems**
  - The saying goes, "Proper Object oriented design makes a developer's life **easy**, whereas bad design makes it a **disaster**."

- **What is Object Oriented Design Principles?**
  - The **process of planning** software system using some **guideline or principles** where object will interact **with best possible way**.
  - Benefit
    - It will make developer **life easy**.
    - It will make software more **manageable**.

# SOLID Object Oriented Design Principles

- Introduced by by **Mr. Robert Martin** (commonly known as Uncle Bob)

- Acronyms of five principles
  - **S-SRP** - Single responsibility Principle
  - **O-OCP** - Open-closed Principle
  - **L-LSP** - Liskov substitution Principle
  - **I-ISP** - Interface segregation Principle
  - **D-DIP** - Dependency inversion Principle

# More Principles

- More principles other than those categorized by Uncle Bob
  - Program to Interface Not Implementation.
  - Don't Repeat Yourself.
  - Encapsulate What Varies.
  - Depend on Abstractions, Not Concrete classes.
  - Least Knowledge Principle.
  - Favor Composition over Inheritance.
  - Hollywood Principle.
  - Apply Design Pattern wherever possible.
  - Strive for Loosely Coupled System.
  - Keep it Simple and Sweet / Stupid. (KISS)

# SRP– Single Responsibility Principle

# SRP – Single Responsibility Principle

The Single Principle states that

***Every object should have a single responsibility and that responsibility should be entirely encapsulated by the class. - Wikipedia***

***There should never be one reason for a class to change. – Robert C. "Uncle Bob" Martin***

# Real World Example

S/W engineer

Author

News paper editor

CAN'T YOU DO ANYTHING RIGHT?
GLASBERGEN

Copyright 2003 Randy Glasbergen. www.glasbergen.com

# Cohesion and Coupling

- **Cohesion**
  - **How** closely **related methods and class level variables** are in a class.
  - Or, How strongly related or focused are various responsibilities of a module

- **Coupling**
  - The notion of coupling attempts to capture this concept of "**how strongly**" **different modules** are **interconnected**
  - Or, the degree to which each program module relies on each one of the other module

**Strive for low Coupling and High Cohesion!**

# Responsibilities are Axes of Change

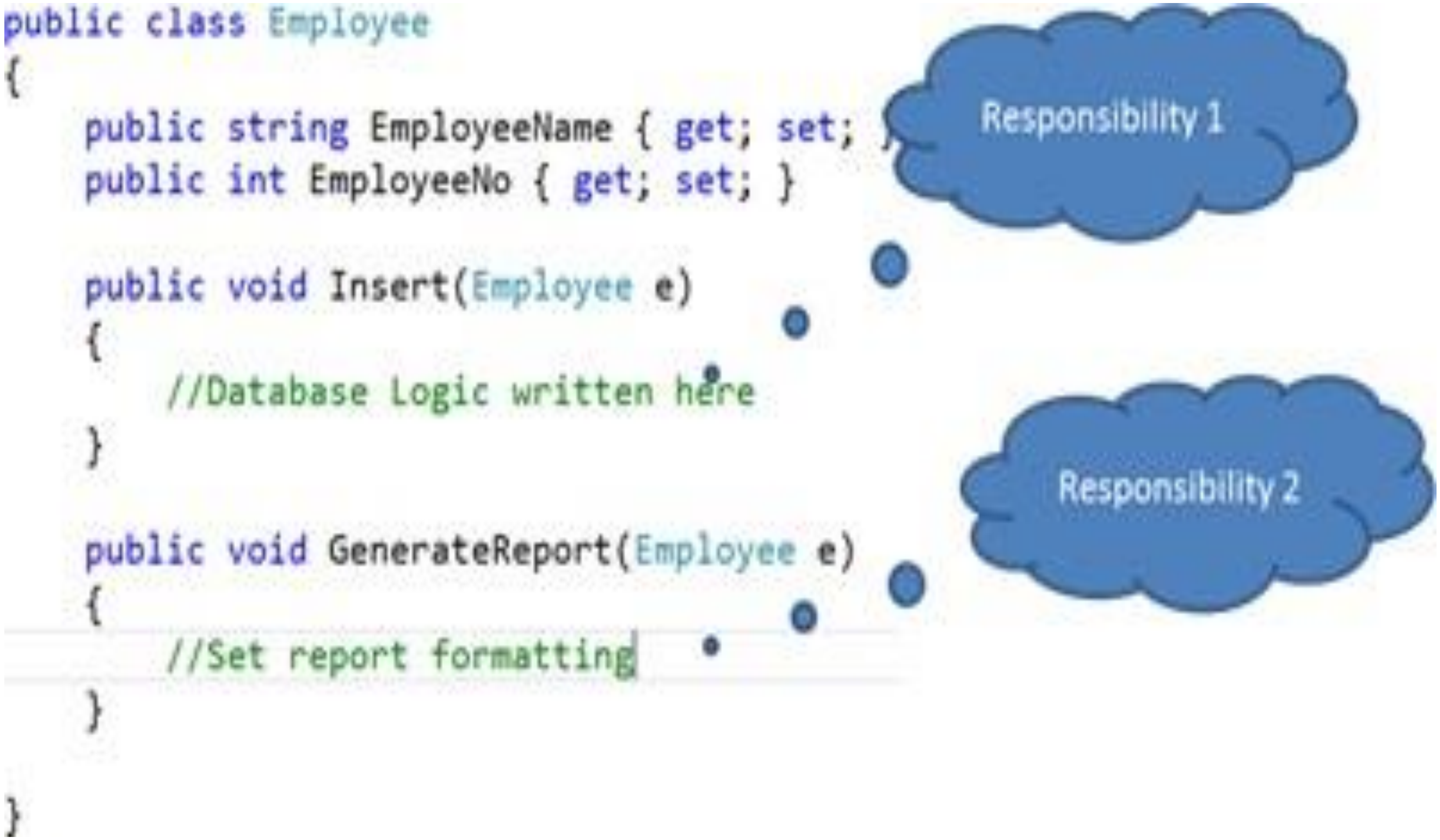
- Requirements **changes** typically **map** to **responsibilities**

- **More responsibilities == More** likelihood of **change**

- Having **multiple responsibilities** within a class **couples together** these responsibilities

- The **more classes a change** affects, the more likely the change will introduce **errors**.

# Have a look on the following class !



```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }

    public void Insert(Employee e)
    {
        //Database Logic written here
    }

    public void GenerateReport(Employee e)
    {
        //Set report formatting
    }

}
```

Responsibility 1

Responsibility 2

# Demo of SRP

# Solutions which will not Violate SRP

- Now it’s up to us how we achieve this.
- One thing we can do is create three different classes
    - **Employee** – Contains Properties (Data)
    - **EmployeeDB** – Does database operations
    - **EmplyeeReport** – Does report related tasks

# Solutions which will not Violate SRP …

```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }
}
public class EmployeeDB
{
    public void Insert(Employee e)
    {
        //Database Logic written here
    }
    public Employee Select()
    {
        return new Employee();
        //Database Logic written here
    }
}
public class EmployeeReport
{
    public void GenerateReport(Employee e)
    {
        //Set report formatting
    }
}
```

# Can a single class can have multiple methods?

Yes,

A class may have **more than one method.**

A method will have **single responsibility**.

# Summery

- **“a reason to change”**
- Multiple small interfaces (follow **ISP**) can help to **achieve SRP**
- **Following** SRP leads to **lower coupling and higher cohesion**
- Many small classes with distinct responsibilities result in a more **flexible design**

# OCP – Open Close Principle

# OCP – Open Close Principle

The Open Close Principle states that

***Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification - Wikipedia***

*First introduced by Betrand Meyer in 1988*

# Real world example

# What is OCP?

- **Open for extension**
  - Its behavior can be extended to **accommodate new demand.**

- **Close for modification**
  - The existing source code of the module is **not changed or minimum change** when making **enhancement**

# Change behavior without changing code?

- Rely on **abstractions**
- **No limit** to variety of **implementations** of each abstraction
- In **.NET, abstractions** include:
  - Interfaces
  - Abstract Base Classes
- In **procedural** code, some level of OCP can be achieved via **parameters**

# The Problem

- **Adding new rules** require **changes** to every time
- **Each change** can introduce **bugs** and requires **re-testing**, etc.
- We want to **avoid** introducing **changes** that **cascade** through many modules in our application

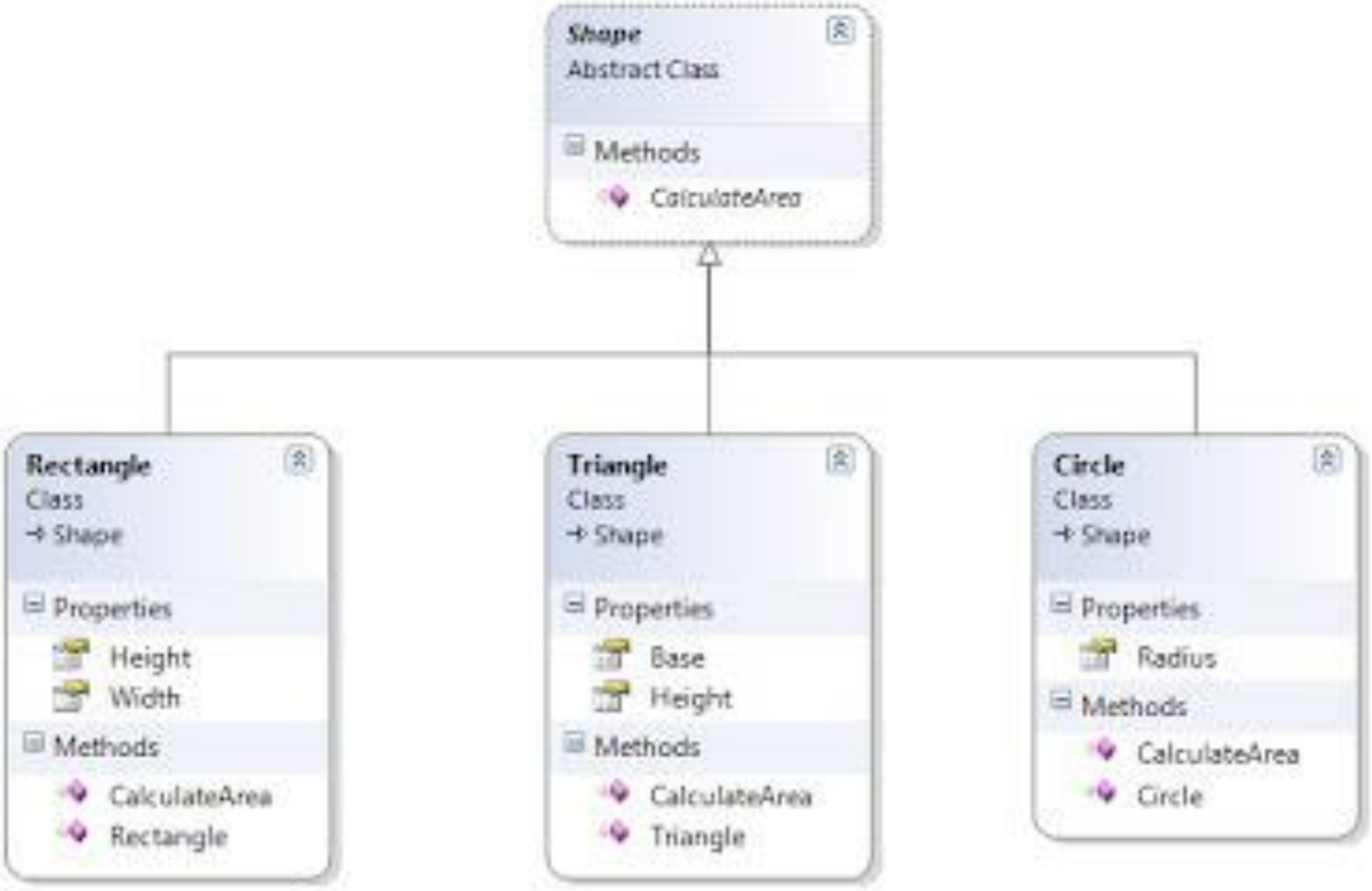
- **Writing new classes** is less likely to introduce problems
  - **Nothing depends** on **new classes** (yet)
  - **New classes have no legacy coupling** to make them hard to design or test

# Three Approaches to Achieve OCP

- **Parameters (Procedural Programming)**
  - Allow client to control behavior specifics via a parameter
  - Combined with delegates/lambda, can be very powerful approach
- **Inheritance / Template Method Pattern**
  - Child types override behavior of a base class (or interface)
- **Composition / Strategy Pattern**
  - Client code depends on abstraction
  - Provides a “plug in” model
  - Implementations utilize Inheritance; Client utilizes Composition

# Demo of OCP

# Implementation of OCP



# OCP Implementation

```
public abstract class Shape
{
    public abstract double CalculateArea();
}

public class Rectangle : Shape
{
    public double Height { get; set; }
    public double Width { get; set; }

    public Rectangle(double height, double width)
    {
        this.Height = height;
        this.Width = width;
    }

    public override double CalculateArea()
    {
        return Height * Width;
    }
}
```

# OCP Implementation …

```
public class Triangle : Shape
 {
     public double Base { get; set; }
     public double Height { get; set; }
     public Triangle(double vbase, double vheight)
     {
         this.Base = vbase;
         this.Height = vheight;
     }
     public override double CalculateArea()
     {
         return 1 / 2.0 * Base * Height;
     }
 }

 public class Circle : Shape
 {
     public double Radius { get; set; }
     public Circle(double radius)
     {
         this.Radius = radius;
     }
     public override double CalculateArea()
     {
         return Math.PI * Radius * Radius;
     }
 }
```

# When do you apply OCP?

- **Experience Tells You**
  - If you know from your own experience in the problem domain that a particular class of **change is likely to recur**, **you can apply OCP** up front in your design

- Don't apply OCP at first
- If the module changes **once**, **accept** it.
- If it changes a **second time**, **refactor** to achieve OCP

# Summery

- OCP yields **flexibility**, **reusability**, and **maintainability**

- **Know** which changes to guard against, and **resist premature abstraction**

# LSP – Liskov Substitution Principle

# LSP – Liskov Substitution Principle

*The Liskov Substitution Principle states that*

***Subtypes must be substitutable for their base types.***

*- Agile Principles, Patterns, and Practices in C#*

*Named for Barbara Liskov, who first described the principle in 1988*

# Real World Example

# Substitutability

- Child classes must not:
  - Remove base class behavior
  - Violate base class invariants

- In general must not require calling code to **know** they are **different from their base type**

# Demo of LSP

# Implementation of LSP

```
public abstract class Shape
   {
       public abstract int Area();

   }

   public class Squre : Shape
   {
       public int SideLength;
       public override int Area()
       {
           return SideLength * SideLength;
       }
   }

   public class Rectangle : Shape
   {
       public int Height { get; set; }
       public int Width { get; set; }

       public override int Area()
       {
           return Height * Width;
       }
   }
```

# Implementation of LSP…

## Testing LSP

```
[TestMethod]
 public void SixFor2x3Rectange()
 {
     var myRectangle = new Rectangle { Height = 2, Width = 3 };
     Assert.AreEqual(6, myRectangle.Area());
 }

 [TestMethod]
 public void NineFor3x3Squre()
 {
     var squre = new Squre { SideLength = 3 };
     Assert.AreEqual(9, squre.Area());
 }

 [TestMethod]
 public void TwentyFor4x5ShapeAnd9For3x3Squre()
 {
     var shapes = new List<Shape>
     {
         new Rectangle{Height = 4, Width = 5},
         new Squre{SideLength = 3}
     };
     var areas = new List<int>();
     #region problem
     //So you are following both polymorphism and OCP
     #endregion
     foreach (Shape shape in shapes)
     {
         areas.Add(shape.Area());
     }
     Assert.AreEqual(20, areas[0]);
     Assert.AreEqual(9, areas[1]);
 }
```

# Summery

- **Extension** of open close principle
- LSP allows for **proper use of polymorphism**
- Produces more **maintainable** code
- Remember **IS-SUBSTITUTABLE-FOR** instead of **IS-A**

# ISP– Interface Segregation principle

# ISP– Interface Segregation principle

*The Interface Segregation Principle states that*

***Clients should not be forced to depend on the methods they do not use.***

***Corollary : Prefer small, cohesive interface to fat interfaces***

# ISP - Real world example

# What is ISP?

- **Many client specific interfaces are better** than one general purpose interface
- The dependency of one **class** to another one should **depend on the smallest possible interface**
- In simple words, **if your interface is fat, break it into multiple interfaces.**

# Demo of ISP

# ISP Violation

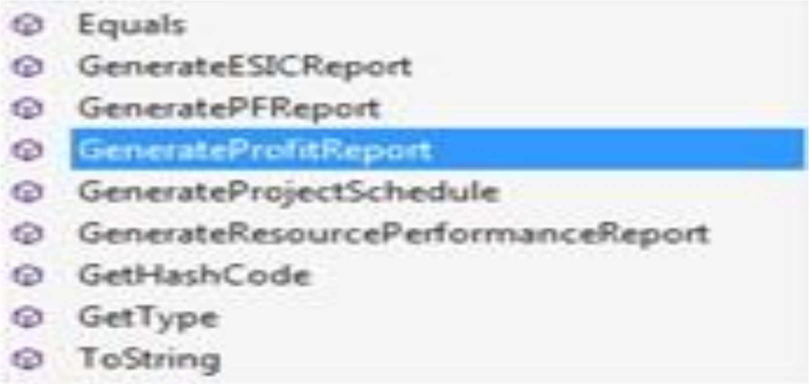
```
public interface IReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();

    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();

    void GenerateProfitReport();
}
public class ReportBAL : IReportBAL
{
    public void GeneratePFReport()
    {/*...............*/}
    public void GenerateESICReport()
    {/*...............*/}
    public void GenerateResourcePerformanceReport()
    {/*...............*/}
    public void GenerateProjectSchedule()
    {/*...............*/}
    public void GenerateProfitReport()
    {/*...............*/}
}
```

# ISP Violation …

```
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 ();
    }
}
```

# ISP Violation …

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



```
Equals
GenerateESICReport
GeneratePFReport
GenerateProfitReport
GenerateProjectSchedule
GenerateResourcePerformanceReport
GetHashCode
GetType
ToString
```

# Refactoring code following ISP

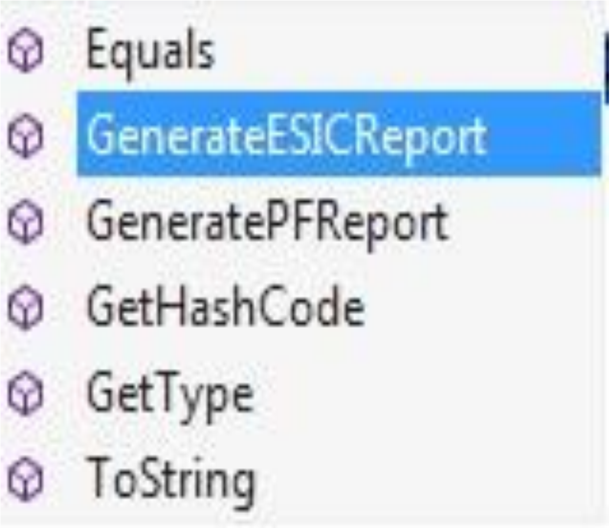
```
public interface IEmployeeReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();
}
public interface IManagerReportBAL : IEmployeeReportBAL
{
    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();
}
public interface IAdminReportBAL : IManagerReportBAL
{
    void GenerateProfitReport();
}
```

# Refactoring code following ISP …

```
public class ReportBAL : IAdminReportBAL
{
    public void GeneratePFReport()
    {/*...............*/}

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

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

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

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

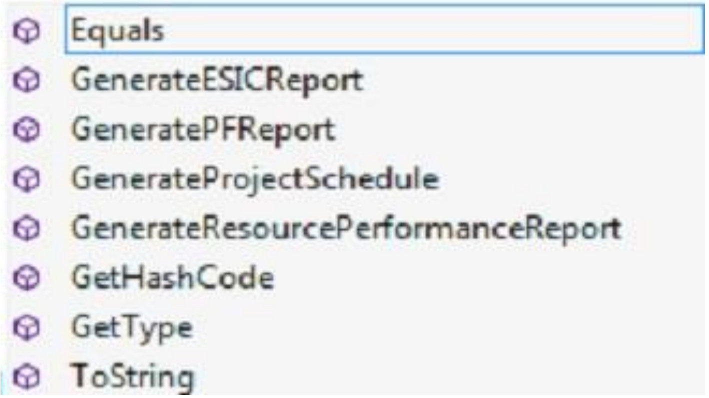
# Refactoring code following ISP …

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



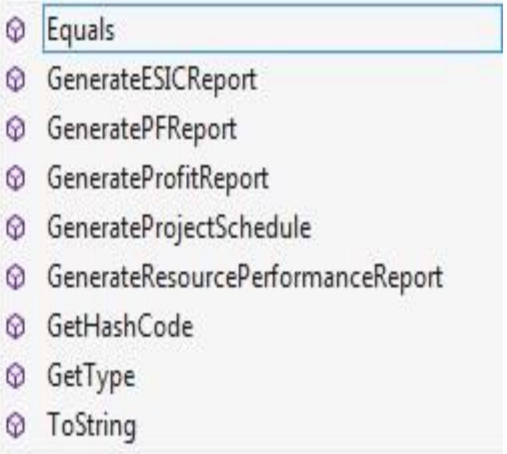
# Refactoring code following ISP …

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



# Refactoring code following ISP …

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



# DIP – Dependency Inversion principle

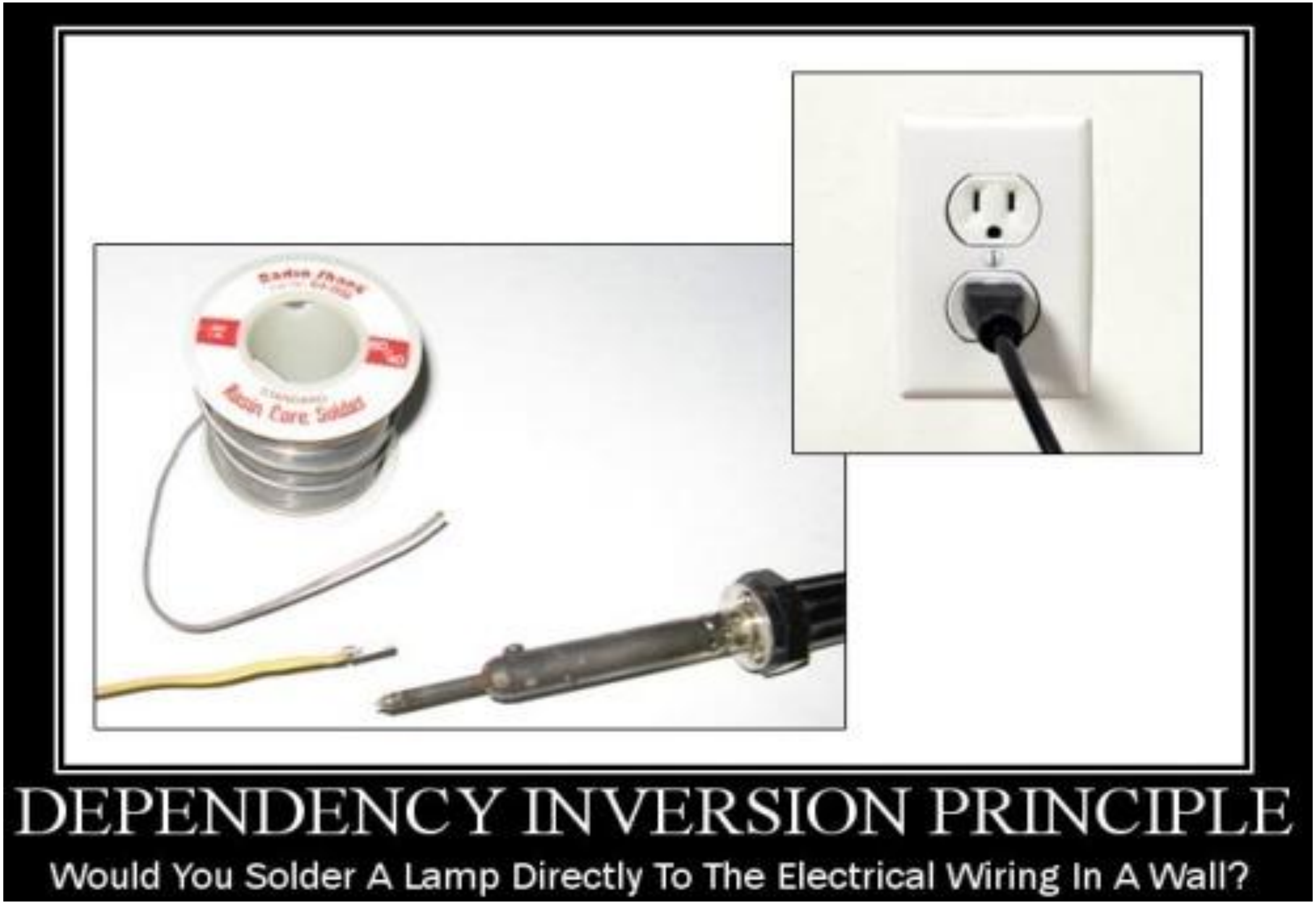
# DIP – Dependency Inversion principle

*The Dependency Inversion Principle states that*

***"High level modules should not depend upon low level modules. Both should depend on abstractions."***

*Abstraction should not depends on details. Details should depend on abstractions.*

# DIP – Dependency Inversion principle


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

# What are dependencies?

- Framework
- Third party libraries
- Database
- File System
- Email
- The new keyword
- System resources (clock) etc.

# Traditional Programming and dependencies

- High level module call low level module

- Use interface depends on
  - Business logic depends on
    - Infrastructure
    - Utility
    - Data access

- Static method are used for convenience  or as façade layer
- Class instantiation / call stack logic is scattered through out all modules
  - Violation of Single Responsibilities principle

# Problem for dependencies

- Tight coupling
- No way to change implementation details
  - OCP Violation
- Difficult to test

# Solution : Dependency Injection

- Dependency Injection is a technique that is used to allow calling code to inject the dependencies a class needs when it is instantiated.
- The Hollywood Principle
  - "Don't call us; we'll call you"
- Three Primary Techniques
  - Constructor Injection
  - Property Injection
  - Parameter Injection
- Other methods exist as well

# Constructor Injection

- Dependencies are passed in via constructor
- Pros
  - Classes self-document what they need to perform their work
  - Works well with or without a container
  - Classes are always in a valid state once constructed
- Cons
  - Constructors can have many parameters/dependencies (design smell)
  - Some features (e.g. Serialization) may require a default constructor
  - Some methods in the class may not require things other methods require (design smell)

# Property Injection

- Dependencies are passed in via a property
  - Also known as “setter injection”

- Pros
  - Dependency can be changed at any time during object lifetime
  - Very flexible

- Cons
  - Objects may be in an invalid state between construction and setting of dependencies via setters
  - Less intuitive

# Parameter Injection

- Dependencies are passed in via a method parameter

- Pros
  - Most granular
  - Very flexible
  - Requires no change to rest of class
- Cons
  - Breaks method signature
  - Can result in many parameters (design smell)

- Consider if only one method has the dependency, otherwise prefer constructor injection

# Constructor Injection

```
 public class OnlineOrder : Order
 {
        private readonly INotificationService _notificationService;
        private readonly PaymentDetails _paymentDetails;
        private readonly IPaymentProcessor _paymentProcessor;
        private readonly IReservationService _reservationService;

        public OnlineOrder(Cart cart,
                           PaymentDetails paymentDetails,
                           IPaymentProcessor paymentProcessor,
                           IReservationService reservationService,
                           INotificationService notificationService)
            : base(cart)
        {
            _paymentDetails = paymentDetails;
            _paymentProcessor = paymentProcessor;
            _reservationService = reservationService;
            _notificationService = notificationService;
        }
      //……..
}
```

# Where do we instantiate objects ?

- Applying Dependency Injection typically results in many interfaces that eventually need to be instantiated somewhere… but where?
- Default Constructor
  - You can provide a default constructor that news up the instances you expect to typically need in your application
  - Referred to as “poor man’s dependency injection” or “poor man’s IoC”
- Main
  - You can manually instantiate whatever is needed in your application’s startup routine or main() method
- IoC Container
  - Use an “Inversion of Control” Container

# IOC Container

- Responsible for object graph instantiation
- Initiated at application startup via code or configuration
- Managed interfaces and the implementation to be used are Registered with the container
- Dependencies on interfaces are Resolved at application startup or runtime

# Example IOC Container in .NET

- Microsoft Unity
- StructureMap
- Ninject
- Windsor
- Funq / Munq

Thank you

# Modification History

| SL | Version | Modification Description | Update date |
|---|---|---|---|
| 1 | 1.0 | Initial creation | 21/02/2014 |
| | | | |

# References

- http://courses.cs.washington.edu/courses/cse403/96sp/coupling-cohesion.html
- http://en.wikipedia.org/wiki/Coupling_(computer_programming)