ICT365

Software Development Frameworks

Dr Afaq Shah

Murdoch
UNIVERSITY

Design Principles

Murdoch
UNIVERSITY

In this Topic

Murdoch UNIVERSITY

SOLID Design Principles

S Single Responsibility Principle

O Open Closed Principle

L Liskovs Substitution Principle

I Interface Segregation Principle

D Dependency Inversion principle

Bad designs and poor code is not good because it is hard to change. Bad designs are:

- Rigid (change affects too many parts of the system)
- Fragile (every change breaks something unexpected)
- Immobile (impossible to reuse)

Single Responsibility Principle

Murdoch UNIVERSITY

"A class should have one reason to change"

Meaning

There can be only one requirement that when changed, will cause a class to change.

How to do this ?

breaking your story/requirement/code to smaller chunks/class in such a way that all common behavior is grouped together under one component.

Single Responsibility means your class has to be very focused

Outcome

You will end up having multiple classes, each related to one specific behavior. Each of these classes can now be used as a component.

Single Responsibility Principle

Murdoch UNIVERSITY

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 more than one reason for a class to change. – Robert Martin

Cohesion and Coupling

Murdoch UNIVERSITY

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

Murdoch UNIVERSITY

- 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.

Murdoch
UNIVERSITY

```
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

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

EmployeeReport – Does report related tasks

```
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 have multiple methods?

Murdoch UNIVERSITY

Yes,

A class may have more than one method.

A method will have single responsibility.

Summary

Murdoch University

- "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

Open Closed Principle

Murdoch UNIVERSITY

First introduced by Betrand Meyer in 1988

"Your class/module/contracts should be open for extension and closed for modification"

Meaning

You should be able to extend the behavior of a class without changing it.

Once a class is done, it should not be modified.

How to do this ?

Creating new class by using inheritance

or

abstraction

Example

Power socket/adapter can be modified but easily extended using an extension

What is OCP?

Murdoch UNIVERSITY

- 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

Open Closed Principle

Murdoch University

To implement OCP, we should be SRP

Output

Lots of components(class) for same behavior but different implementation

The Problem

Murdoch UNIVERSITY

- Adding new rules require changes to classes 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

Shape
Abstract Class
Methods
CalculateArea

Rectangle
Class
→ Shape
Properties
Height
Width
Methods
CalculateArea
Rectangle

Triangle
Class
→ Shape
Properties
Base
Height
Methods
CalculateArea
Triangle

Circle
Class
→ Shape
Properties
Radius
Methods
CalculateArea
Circle

```
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;
    }
}
```

```
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?

Murdoch UNIVERSITY

- Experience Tells You
- Don’t apply OCP at first
- If the module changes once, accept it.
- If it changes a second time, refactor to achieve OCP

Summary

Murdoch University

- OCP yields flexibility, reusability, and maintainability
- Know which changes to guard against, and resist premature abstraction

Liskov Substitution Principle

Murdoch University

Subtypes must be substitutable for their base types.

Barbara Liskov first described the principle in 1988

“If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.”

Meaning

We should be able to substitute every derived class for their parent class. A subclass (outcome of OCP) should behave in such a way that it will not cause problem when used instead of its super class (outcome of SRP)

LSP

Parent

Child

Substitution

Violation

Parent

Child

Substitutability

Murdoch UNIVERSITY

- 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

Summary

Murdoch UNIVERSITY

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

Interface Segregation Principle

Murdoch University

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

How to do this?

Prefer small, cohesive interface to fat interfaces

Output

Smaller Interfaces with specific responsibility.

Real life ISP

ISP..

Murdoch UNIVERSITY

- 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.

ISP Violation

Murdoch UNIVERSITY

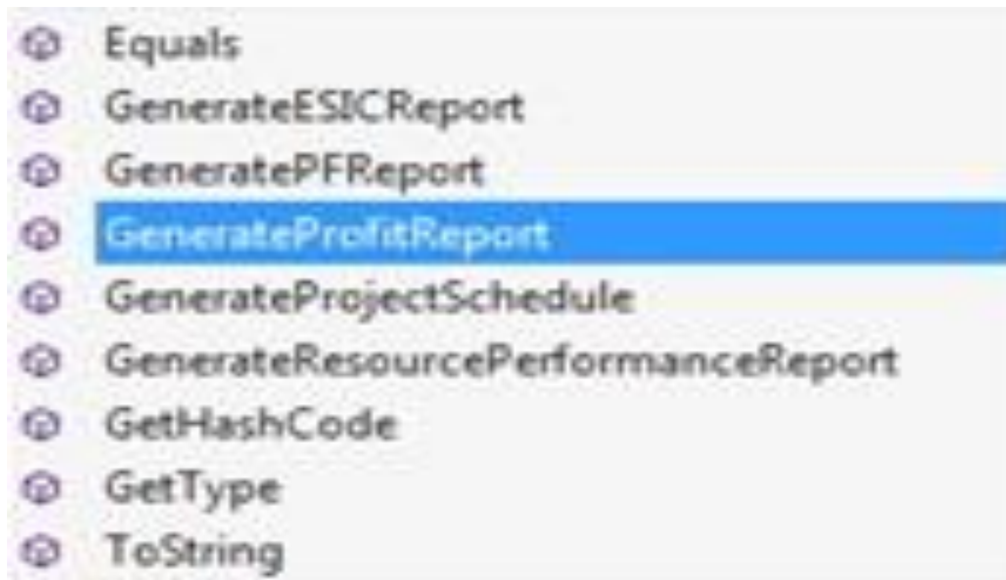
```
public interface IReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();

    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()
    {/*...............*/}
}
```

```
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();
    }
}
```



Refactoring code following ISP

Murdoch University

```
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()
    {/*...............*/}
}
```

```
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();
    }
}
```

Dependency Inversion Principle

Murdoch UNIVERSITY

This principle is about dependencies among the components (such as two modules, two classes) of the software.

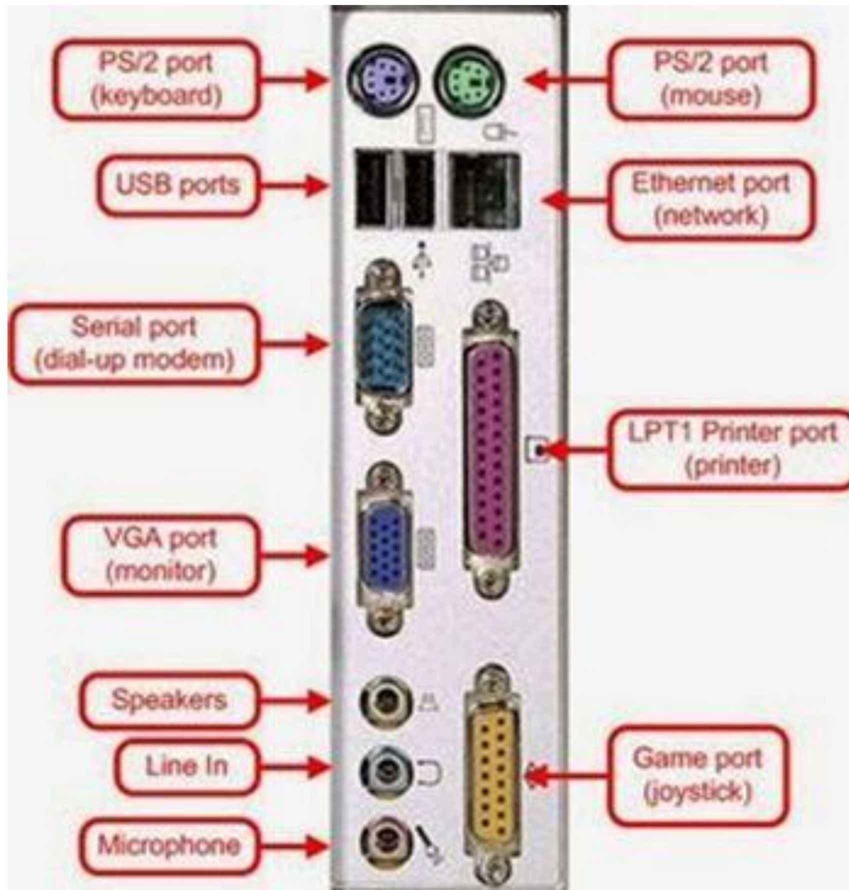
“A high level module should not depend on a low level module. Both should depend on abstraction.

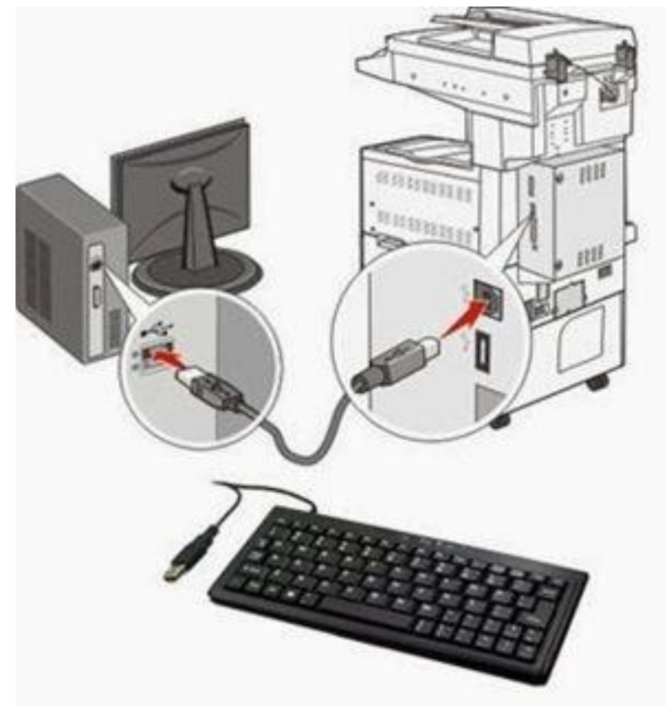
Abstraction should not depend upon details. Details should depend upon abstraction.”

Meaning

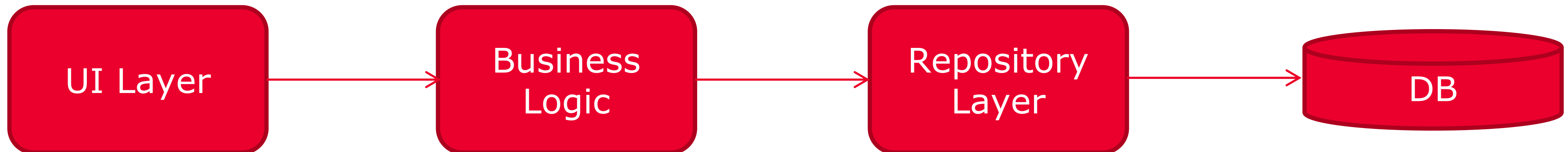
Your classes should not depend on specific implementation, avoid instantiation of a class X inside another Y, as this makes your class Y directly dependent on X and any change in X might break Y.

DIP





Dependency Inversion Principle



We should not use NEW keyword inside a class as this creates a dependency.

How to do this?

By using Interfaces or Abstract classes to create generic types.

By not using NEW keyword to instantiate an object and instead inject the dependencies using constructor

If NEW is not to be used, then who will create required object?

By having a Abstract Factory Method responsible for create of new object.

What are dependencies?

Murdoch UNIVERSITY

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

Dependencies Problem

Murdoch UNIVERSITY

- Tight coupling
- No way to change implementation details

  OCP Violation
- Difficult to test

Remember..

Murdoch UNIVERSITY

- Try avoiding over engineering and keep the solution simple and readable.
- Decide how far you want to go with SOLID, as splitting requirement to SOLID principles is debatable.

Summary

Murdoch UNIVERSITY

- **SOLID Principles are principles of class design.**
- **S**RP: Single Responsibility Principle

  An object should have only a single responsibility & all the responsibility should be entirely encapsulated by the class.

  There should never be more than one reason for a class to change
- **O**CP: Open/Closed Principle

  Software entities should be open for extension, but closed for modification
- **L**SP: Liskov Substitution Principle

  Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program
- **I**SP: Interface Segregation Principle

  many client specific interfaces are better than one general purpose interface

  once an interface has gotten too 'fat' split it into smaller and more specific interfaces so that any clients of the interface will only know about the methods that pertain to them. No client should be forced to depend on methods it does not use
- **D**IP: Dependency Inversion Principle

  Depend upon Abstractions. Do not depend upon concretions.

  Dependency Injection (DI) is one method of following this principle.

More principles

Murdoch UNIVERSITY

Program to Interface Not Implementation.

Don't Repeat Yourself.

Encapsulate What Varies.

Depend on Abstractions, Not Concrete classes.

Apply Design Pattern wherever possible.

Strive for Loosely Coupled System.

Keep it Simple and Sweet / Stupid. (KISS)

Least Knowledge Principle.

Hollywood Principle.

SOLID

Software development is not a Jenga game.

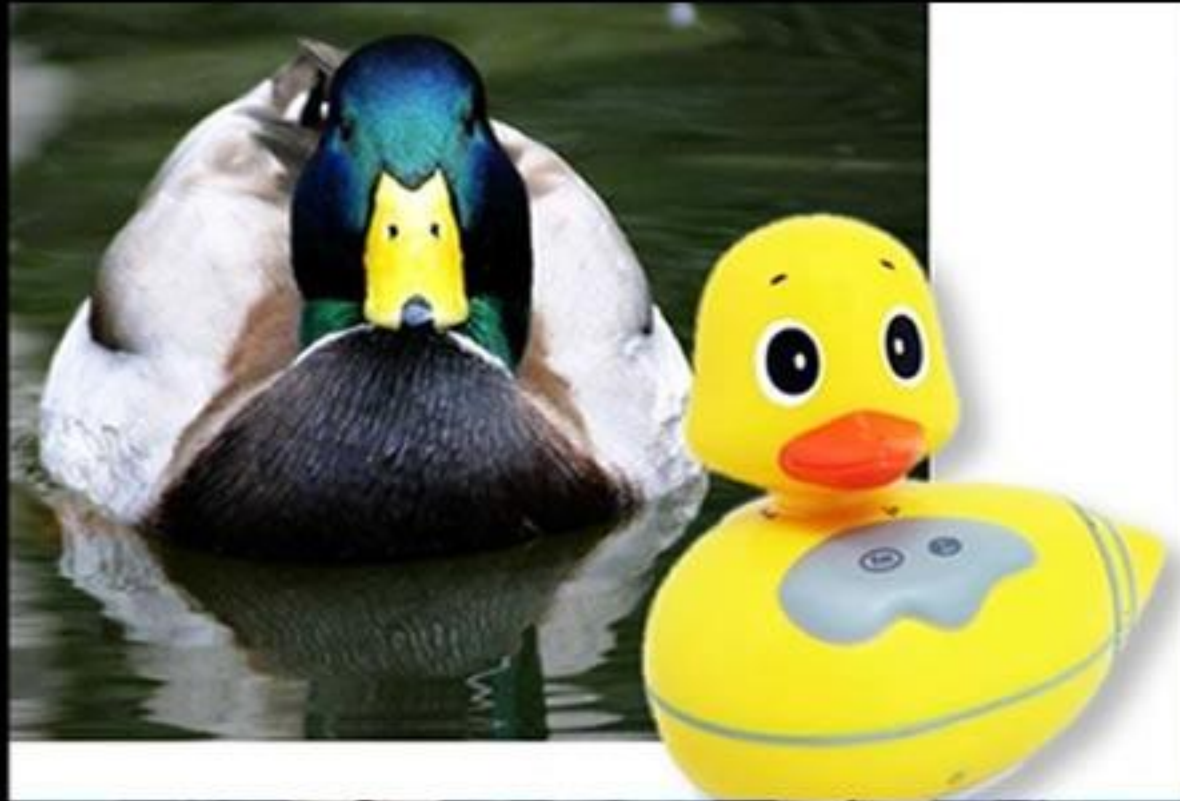
Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

Open-Closed Principle

Open-chest surgery isn't needed when putting on a coat.

Murdoch
UNIVERSITY


Liskov Substitution Principle
If it looks like a duck and quacks like a duck but needs batteries,
you probably have the wrong abstraction.

Interface Segregation Principle

You want me to plug this in *where?*

Dependency Inversion Principle

Would you solder a lamp directly
to the electrical wiring in a wall?

LSP Example

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

    }

    public class Square : 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;
        }
    }
```

Murdoch
UNIVERSITY

```
[TestMethod]
public void SixFor2x3Rectangle()
{
    var myRectangle = new Rectangle { Height = 2, Width = 3 };
    Assert.AreEqual(6, myRectangle.Area());
}
[TestMethod]
public void NineFor3x3Square()
{
    var square = new Square { SideLength = 3 };
    Assert.AreEqual(9, sqaure.Area());
}
```

```
[[TestMethod]
    public void TwentyFor4x5ShapeAnd9For3x3Squre()
    {
        var shapes = new List<Shape>
        {
            new Rectangle{Height = 4, Width = 5},
            new Square{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]);
    }
```

C# Reference

- **Design Fundamentals, Chapter 10: Component Guidelines**
- https://msdn.microsoft.com/en-au/library/ee658121.aspx
- **Sample chapter: The Liskov Substitution Principle**
- https://blogs.msdn.microsoft.com/microsoft_press/2014/10/31/sample-chapter-the-liskov-substitution-principle/
- **Adaptive Code via C#: Agile coding with design patterns and SOLID principles**
- https://blogs.msdn.microsoft.com/microsoft_press/2014/10/22/new-book-adaptive-code-via-c-agile-coding-with-design-patterns-and-solid-principles/
- https://github.com/garymcleanhall/AdaptiveCode

C# Reference

Murdoch UNIVERSITY

- **OOPS Principles (SOLID Principles)**
- https://code.msdn.microsoft.com/windowsapps/OOPS-Principles-SOLID-7a4e69bf
- **The SOLID Principles, Explained with Motivational Posters**
- https://blogs.msdn.microsoft.com/cdndevs/2009/07/15/the-solid-principles-explained-with-motivational-posters/
- **SOLID architecture principles using simple C# examples**
- http://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp

Murdoch UNIVERSITY

Acknowledgements

Sources used in this presentation include:

The SOLID Principles, Explained with Motivational Posters:

https://blogs.msdn.microsoft.com/cdndevs/2009/07/15/the-solid-principles-explained-with-motivational-posters/

SOLID Software Development by Ken Burkhardt

Object Oriented Design Principles with C# by Md. Mahedee Hasan

Examples from https://www.c-sharpcorner.com