The
S.O.L.I.D. Principles

of Object Oriented Programming

Object-Oriented programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
- Decoupling

S.O.L.I.D.

- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle

- A class should have one and only one reasons to change.
- You had one job!

Single Responsibility Principle

```
public interface IEmployee
    {
        string FirstName { get; set; }
        string LastName { get; set; }
        float HourlyRate { get; set; }
        …

        float CalculatePay(float hoursWorked);
        string ReportHours();
        void Save();
    }
```

Single Responsibility Principle

```
public interface IEmployee
    {
        string FirstName { get; set; }
        string LastName { get; set; }
        float HourlyRate { get; set; }
        …
    }
```

```
public interface IDataAccess
{
  void Save();
}
```

```
public interface IReporting
{
  string ReportHours(Employee employee);
}
```

```
public interface IPayroll
{
    float CalculatePay(
                Employee employee,
                float hoursWorked);
}
```

Open-Closed Principle

- Objects or entities should be
  - open for extension,
  - but closed for modification
- Add new behavior; don't change existing behavior

Open-Closed Principle Strategies

- Parameters
- Inheritance
- Composition / Strategy Pattern

Open-Closed Principle

```
public class Drawing
{
    public void DrawAllShapes(object[] shapes)
    {
        foreach (var shape in shapes)
        {
            if (shape is Circle)
            {
                var circle = (Circle)shape;
                DrawCircle(circle);
            }
            else if (shape is Square)
            {
                var square = (Square)shape;
                DrawSquare(square);
            }
        }
    }
}
```

Open-Closed Principle

```
public interface IShape
{
    void Draw();
}
```

```
public class Circle : IShape
{
    public int Radius { get; set; }
    public int CenterX { get; set; }
    public int CenterY { get; set; }

    public void Draw()
    {
        var center = new Tuple<int, int>
            (this.CenterX, this.CenterY);
        var radius = this.Radius;
    }
}
```

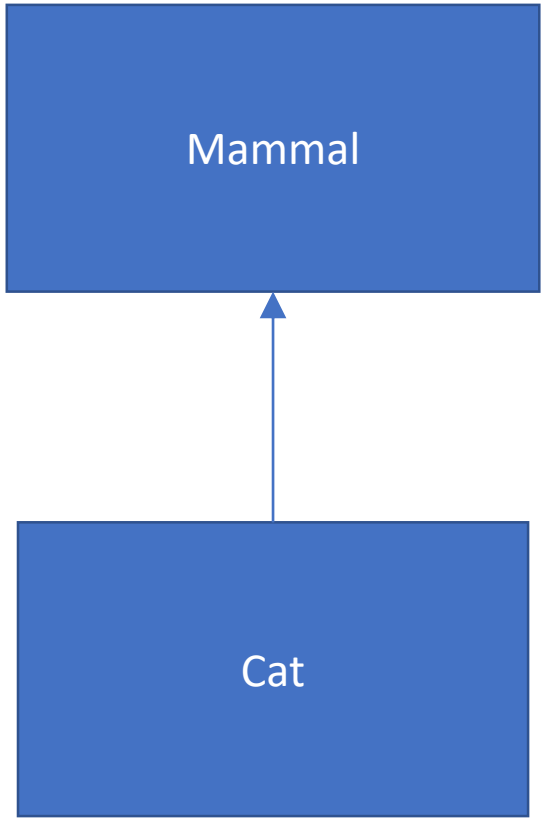
```
public class Square : IShape
{
    public int Side { get; set; }
    public int Top { get; set; }
    public int Left { get; set; }

    public void Draw()
    {
        var topLeft = new Tuple<int, int>
                        (this.Top, this.Left);
        var side = this.Side;
    }
}
```

Liskov Substitution Principle

- If something is true for the base class, it must be true for every derived class

Liskov Substitution Principle

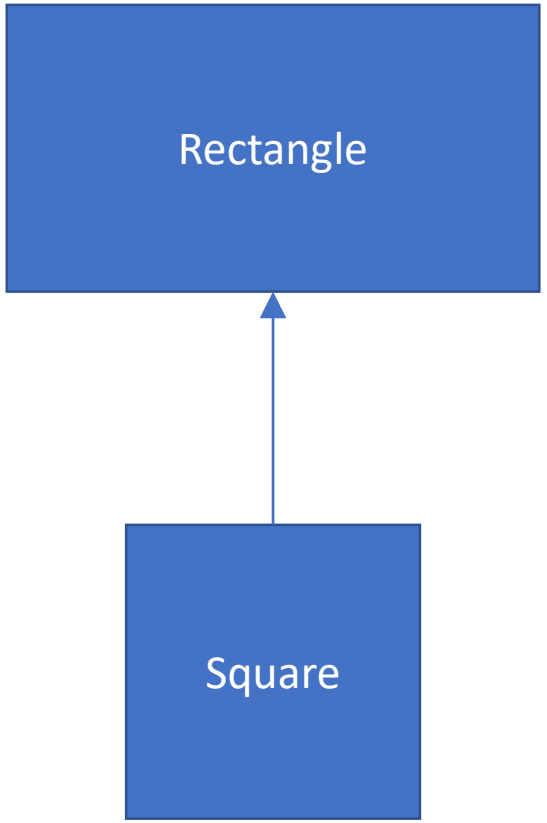


Inheritance

Cat is a Mammal

Cat inherits from Mammal

Liskov Substitution Principle



Inheritance

Square is a Rectangle

Square inherits from Rectangle

Liskov Substitution Principle

```
public class Rectangle
{
    public virtual int Height { get; set; }
    public virtual int Width { get; set; }
}
```

```
Rectangle r = new Square() { Height = 10, Width = 5 };
var area = r.Width * r.Height;
```

```
public class Square : Rectangle
{
    private int _width;
    private int _height;
    public override int Width
    {
        get { return _width; }
        set
        {
            _width = value;
            _height = value;
        }
    }

    public override int Height
    {
        get { return _height; }
        set
        {
            _height = value;
            _width = value;
        }
    }
}
```

Interface Segregation Principle

- A client should never be forced to implement and interface that it doesn't use

Interface Segregation Principle

```
interface IMachine
{
    void Print(List<Document> docs);
    void Staple(List<Document> docs);
    void Fax(List<Document> docs);
    void Scan(List<Document> docs);
    void PhotoCopy(List<Document> docs);
}
```

```
class Machine : IMachine
{
    public void Print(List<Document> docs)
    {
        // Print the items.
    }

    public void Staple(List<Document> docs)
    {
        // Staple the items.
    }

    public void Fax(List<Document> docs)
    {
        // Fax the items.
    }

    public void Scan(List<Document> docs)
    {
        // Scan the items.
    }

    public void PhotoCopy(List<Document> docs)
    {
        // Photocopy the items.
    }
}
```

From: https://www.codeproject.com/Tips/766045/Interface-Segregation-Principle-ISP-of-SOLID-in-Cs

Interface Segregation Principle

```
interface IMachine
{
    void Print(List<Document> docs);
    void Staple(List<Document> docs);
    void Fax(List<Document> docs);
    void Scan(List<Document> docs);
    void PhotoCopy(List<Document> docs);
}
```

```
class Printer : IMachine
{
    public void Print(List<Document> docs)
    {
        // Print the items.
    }

    public void Staple(List<Document> docs)
    {
      throw new NotImplementedException();
    }

    public void Fax(List<Document> docs)
    {
      throw new NotImplementedException();
    }

    public void Scan(List<Document> docs)
    {
        throw new NotImplementedException();.
    }
    public void PhotoCopy(List<Document> docs)
    {
        throw new NotImplementedException();
    }
}
```

Interface Segregation Principle

```
interface IPrinter
{
    void Print(List<Document> docs);
}
```

```
interface IScanner
{
    void Scan(List<Document> docs);
}
```

```
interface IStapler
{
    void Staple(List<Document> docs);
}
```

```
interface IFax
{
    void Fax(List<Document> docs);
}
```

```
interface IPhotocopier
{
    void PhotoCopy(List<Document> docs);
}
```

```
public class Printer : IPrinter
{
    public void Print(List<Document> docs)
    {
        // Print document
    }
}
```

Interface Segregation Principle

```
interface IPrinter
{
    void Print(List<Document> docs);
}
```

```
interface IScanner
{
    void Scan(List<Document> docs);
}
```

```
interface IStapler
{
    void Staple(List<Document> docs);
}
```

```
interface IFax
{
    void Fax(List<Document> docs);
}
```

```
interface IPhotocopier
{
    void PhotoCopy(List<Document> docs);
}
```

```
public class PrinterScannerCopier : IPrinter, IScanner, IPhotocopier
{
    public void PhotoCopy(List<Document> docs)
    {
        // Photocopy documents;
    }

    public void Print(List<Document> docs)
    {
        // Print documents
    }

    public void Scan(List<Document> docs)
    {
        // Scan documents
    }
}
```

Dependency Inversion Principle

- Entities must depend on abstractions, not on concrete implementations
- Decoupling
- High level module
- Don't directly create new concreate classes within the body of your class; Let something else control creation of concrete classes

- 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

Dependency Inversion Principle

References

- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod