C# : Object Oriented Design

Modern OOD

OdeToCode.com

Overview

- Single Responsibility Principle
- Open-closed principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Injection Principle

Single Responsibility Principle

A class should have only one reason to change

- Requirements
  - Create a class to aggregate measurements
  - Can select grouping size (groups of 4, groups of 2)
  - Can select aggregating function (average, mean)

```
public class Measurement
{
    public double HighValue { get; set; }
    public double LowValue { get; set; }
}
```

Single Responsibility Principle

- One solution
  - Create aggregator class
  - Pass group size as integer
  - Use enum to specify algorithm

```
var aggregator = new MeasurementAggregator(_data);
var result = aggregator.Aggregate(2, AggregationType.Mean);
```

```
public IEnumerable<Measurement> Aggregate(
             int groupingSize, AggregationType type)
{

}
```

Single Responsibility Principle

- **Problems**
  - Aggregator responsible for calculations, grouping, results
  - Difficult to change
  - Primitive obsession

```
private Measurement Aggregate(IEnumerable<Measurement> measurements,
                             AggregationType type)
{
    Measurement result = null;
    switch(type)
    {
        case AggregationType.Mean:
                result = Average(measurements);
            break;
        case AggregationType.Mode:
                result = Mode(measurements);
            break;

    }
    return result;
}
```

Single Responsibility Principle

- Another solution
  - *Smaller* classes with well defined *roles and responsibilities*
  - Aggregator class only orchestrates details

```
var aggregator = new MeasurementAggregator2(_data);
var result = aggregator.Aggregate(new SizeGrouper(2),
                                  new AveragingCalculator());
```

```
public IEnumerable<Measurement> Aggregate(IGrouper grouper, IAggregateCalculation calculator)
{
    var partitions = grouper.Group(_measurements);
    foreach (var partition   in partitions)
    {
        yield return calculator.Aggregate(partition);
    }
}
```

Open/Closed Principle

Classes should be open for extension but closed for modification

- Remember the first aggregator?
- New grouping algorithms require a modification
- New aggregating algorithms require a modification

```
switch(type)
{
    case AggregationType.Mean:
            result = Average(measurements);
        break;
    case AggregationType.Mode:
            result = Mode(measurements);
        break;

}
```

Open/Closed Principle

- Solutions
  - Introduce new algorithms by writing new classes (strategy pattern)
  - Template method (can be brittle)

```
public class AveragingCalculator : IAggregateCalculation
{
    public Measurement Aggregate(IEnumerable<Measurement> measurements)
    {
        return new Measurement()
        {
            HighValue = measurements.Average(m => m.HighValue),
            LowValue = measurements.Average(m => m.LowValue)
        };
    }
}
```

Liskov Substitution Principle

Subtypes must be substitutable for their base types

```
class LowFrequencyCalculator : IAggregateCalculation
{
    public Measurement Aggregate(IEnumerable<Measurement> measurements)
    {
        if(measurements.Any(m => m.HighValue > 100))
        {
            throw new InvalidOperationException("...");
        }

        // ...
    }
}
```

Liskov Substitution Principle

- Don’t let derived classes cause misbehavior in a base class
- Warning signs on an LSP violation
  - Run time type checking in if/else statements
  - New derived type forces a change in base type

Interface Segregation Principle

Build cohesive abstractions

- Avoid polluting interfaces
  - Separate clients means separate interfaces
  - Don’t force clients to use or implement methods they don’t use
  - Prefer aggregation and delegation over inheritance

Dependency Inversion Principle

Depend only on abstractions, not details

- Isolate business logic from infrastructure details
- Define abstractions to interface between layers

```
case AggregationType.Mean:
        var calc = new AveragingCalculator();
        result = calc.Aggregate(measurements);
        break;
```

☹

Dependency Inversion Principle

- DIP in practice
  - Leads in inversion of control and IoC containers
  - Increased testability & flexibility

Summary - SOLID

- Single Responsibility Principle
- Open-closed principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Injection Principle