|  |
| --- |
|  |
| OOP with C# |
| Applying Object Oriented Design |

# 

# C# OOP

## Objectives

After completing this lab, you should understand how to:

* Break a large class into smaller, focused abstractions
* Apply SOLID principles to C# code

## Overview

In this lab we’ll build a statistics calculator. The calculator aggregates measurements of the following type.

public class Measurement

{

public double HighValue { get; set; }

public double LowValue { get; set; }

}

In our application we’ll be working with a collection of measurement objects. Business requirements say that we’ll need to provide different types of aggregation algorithms (live computing an average, mode, median, and weighted average), across different grouping of measurements (sometimes the business want to aggregate an entire collection of measurements, other times they may want to aggregate pairs, triplets, odd numbered measures, etc).

The solution in the before directory already includes a MeasurementAggregator class and some associated tests. You’ll be adding new features to the MeasurementAggregator and then refactoring to improve the design of the class.

## Part 1 – Implementing the Average

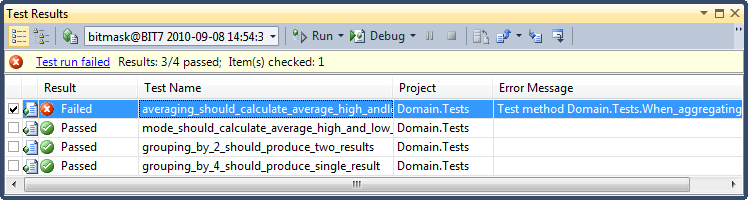
In this first section we’ll be implementing an averaging aggregation.

1. Open the solution file in the before\Statistics folder for this lab.
2. Open the MeasurementAggregatorTests.cs file from the Solution Explorer window (it’s in the Domain.Tests project).

Observe that we have 4 tests in place. Two of the tests ensure the aggregator is grouping properly. The other tests ensure the aggregator can calculate the correct average and the correct mode respectively. Note we are only testing “happy day” scenarios and haven’t focused on boundary conditions (like null objects, empty collections, and irrational numbers).

1. Press Ctrl+R, A to run all the tests.

Only ¾ of the tests should pass. Our first task is to implement the logic to make the failing test pass.



1. Double-click on the failing test.

Inspect the test code. Notice we have a \_data field in the class to hold 4 measurements. We are asking the aggregator to put the measurements in groups of 2, and then produce an average measurement based on each grouping (we should receive 2 “aggregated” measurements as a result). The HighValue and LowValue of an aggregated measurement should represent the mean of the high and low values in each measurement.

Since the test is failing, we need to change the MeasurementAggregator

1. Open the MeasurementAggregator.cs file from the Domain project.

First look at the public Aggregate method. It partitions data, then calls into a private method to perform the aggregation.

Now look at the private Aggregate method. It uses a switch statement to determine the algorithm to apply. Currently the AggregationType.Mean is not handled.

1. In the private Aggregate method, add a case statement to handle AggregationType.Mean. Follow the pattern established for the Mode type (call a method named Mean, passing in measurements and saving the result).

private Measurement Aggregate(IEnumerable<Measurement> measurements,

AggregationType type)

{

Measurement result = null;

switch(type)

{

**case AggregationType.Mean:**

**result = Mean(measurements);**

**break;**

case AggregationType.Mode:

result = Mode(measurements);

break;

}

return result;

}

1. Created the Mean method, including the implementation to calculate the average high and low value.

private Measurement Mean(IEnumerable<Measurement> measurements)

{

return new Measurement()

{

HighValue = measurements.Average(m => m.HighValue),

LowValue = measurements.Average(m => m.LowValue)

};

}

1. Press Ctrl+R, A to run all tests. Proceed to the next section if all your tests are passing.

# Part II – Applying SRP and OCP

*Think about how you felt while working through part I. Was it stressful to change the code in an existing class. Can you imagine working with this class over the next 6 months as new aggregation algorithms are added?*

*Let’s create a second version of the aggregator that applies modern OOP principles.*

1. Add a new class to the Domain project named MeasurementAggregator2.
2. Give the class a constructor that takes an IEnumerable<Measurement> parameter. Save the parameter value into a private field.

public class MeasurementAggregator2

{

public MeasurementAggregator2(

IList<Measurement> measurements)

{

\_measurements = measurements;

}

private IList<Measurement> \_measurements;

}

*We’ve decided the aggregator should only be responsible for facilitating aggregations. All the actual grouping and calculations will be performed by separate components. We’ll create abstractions to represent these components in the following steps.*

1. Add a new interface to the Domain project named IGrouper.
2. Give the interface a method named Group. The method takes an list of measurements as a parameter, and returns a sequence of sequences of measurements.

public interface IGrouper

{

IEnumerable<IEnumerable<Measurement>> Group(

IList<Measurement> measurements);

}

1. Add a new interface to the Domain project named IAggregateCalculator.
2. The calculator interface needs an parameter to take a sequence of measurements and returns a single measurement.

public interface IAggregateCalculator

{

Measurement Aggregate(IEnumerable<Measurement> measurements);

}

1. Return to MeasurementAggregator2. Give the class an Aggregate method that takes a grouper and a calculator. The method returns IEnumerable<Measurement>.
2. Provide an implementation for the Aggregate method.

*The implementation will look like the implementation of the original aggregator, but instead of calling into private methods the implementation will call into the new interfaces.*

public IEnumerable<Measurement> Aggregate(

IGrouper grouper,

IAggregateCalculator calculator)

{

var partitions = grouper.Group(\_measurements);

foreach (var partition in partitions)

{

yield return calculator.Aggregate(partition);

}

}

1. In the test project, open MeasurementAggregatorTests2.cs.
2. Uncomment the class inside (you can highlight lines of code and press Ctrl+K, Ctrl+U to uncomment quickly.

*You’ll notice we have some compiler errors – we don’t have concrete classes built yet.*

1. In the Domain project, create a SizeGrouper class that implements the IGrouper interface. The grouper takes a constructor parameter to indicate the grouping size.
2. Implement the Group method of the new class.

*You can borrow the basic algorithm from the original aggregator class. A final version will look like the following.*

public class SizeGrouper : IGrouper

{

public SizeGrouper(int size)

{

\_size = size;

}

public IEnumerable<IEnumerable<Measurement>> Group(

IList<Measurement> measurements)

{

int total = 0;

while (total < measurements.Count)

{

yield return measurements.Skip(total).Take(\_size);

total += \_size;

}

}

private readonly int \_size;

}

1. In the Domain project, create a ModelCalculator class that implements the IAggregateCalculator interface.

public class ModalCalculator : IAggregateCalculator

{

public Measurement Aggregate(

IEnumerable<Measurement> measurements)

{

var highValue = measurements.GroupBy(m => m.HighValue)

.OrderByDescending(g => g.Count())

.Select(g => g.Key).FirstOrDefault();

var lowValue = measurements.GroupBy(m => m.LowValue)

.OrderByDescending(g => g.Count())

.Select(g => g.Key).FirstOrDefault();

return new Measurement()

{

HighValue = highValue,

LowValue = lowValue

};

}

}

1. Return to MeasurementAggregatorTests2.cs. Comment out all tests but the last test in the class (use Ctrl+K, Ctrl+C to comment selected blocks of code).

*We should have everything implemented for this one test to pass.*

1. Run all tests and ensure everything passes.
2. Uncomment the other three tests.
3. Make all the tests in the project pass by implementing an AveragingCalculator.

*How do you feel about this change compared to the first section?*

## Conclusion

In this module we broke a monolithic class with many responsibiltiies into a series of smaller classes. The original class now has a single responsibility, is closed for modification, and open for extension. You might see now how different pieces of the software are now more reusable, and also more testable (we can test the calculators and groupers independently of the aggregation class).

By praticing these techniques and following the SOLID principles, you can build software that is easier to maintain and extend.