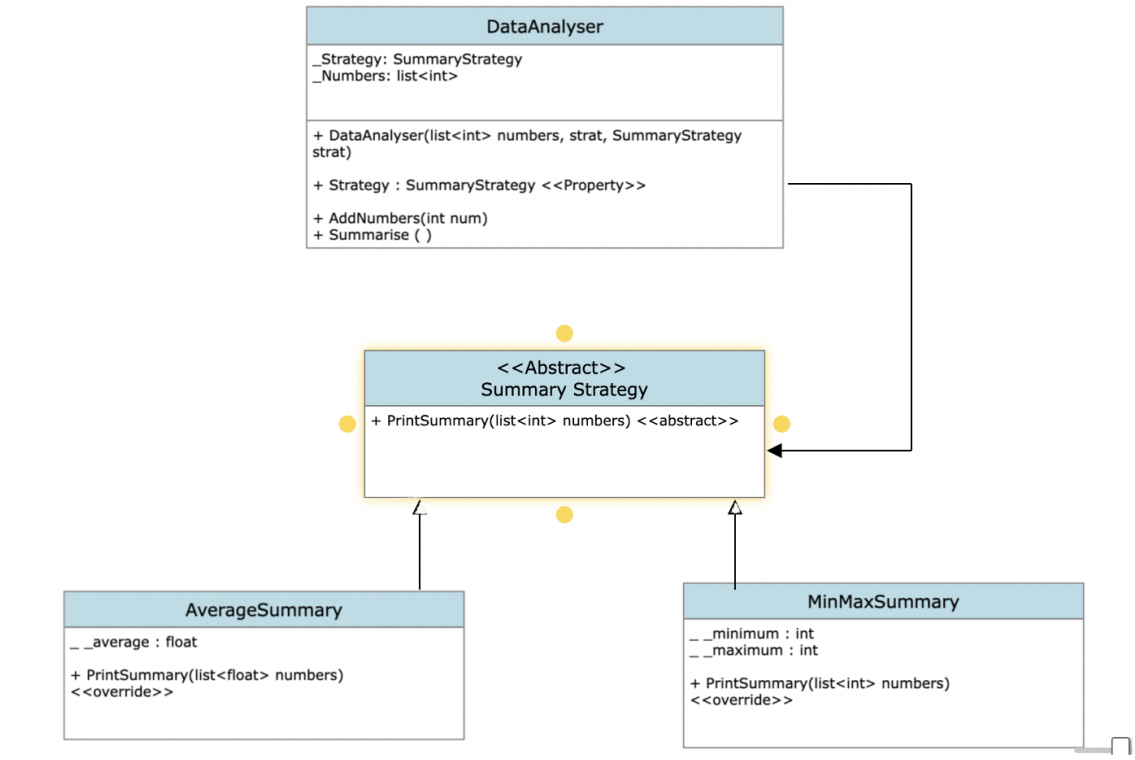
Swinburne University of Technology

COS20007 Object Oriented Programming

Semester Test

PDF generated at 19:39 on Sunday 21st May, 2023


DataAnalyser
_Strategy: SummaryStrategy
_Numbers: list<int>
+ DataAnalyser(list<int> numbers, strat, SummaryStrategy strat)
+ Strategy : SummaryStrategy <<Property>>
+ AddNumbers(int num)
+ Summarise ( )
<<Abstract>>
Summary Strategy
+ PrintSummary(list<int> numbers) <<abstract>>
AverageSummary
_ _average : float
+ PrintSummary(list<float> numbers) <<override>>
MinMaxSummary
_ _minimum : int
_ _maximum : int
+ PrintSummary(list<int> numbers) <<override>>

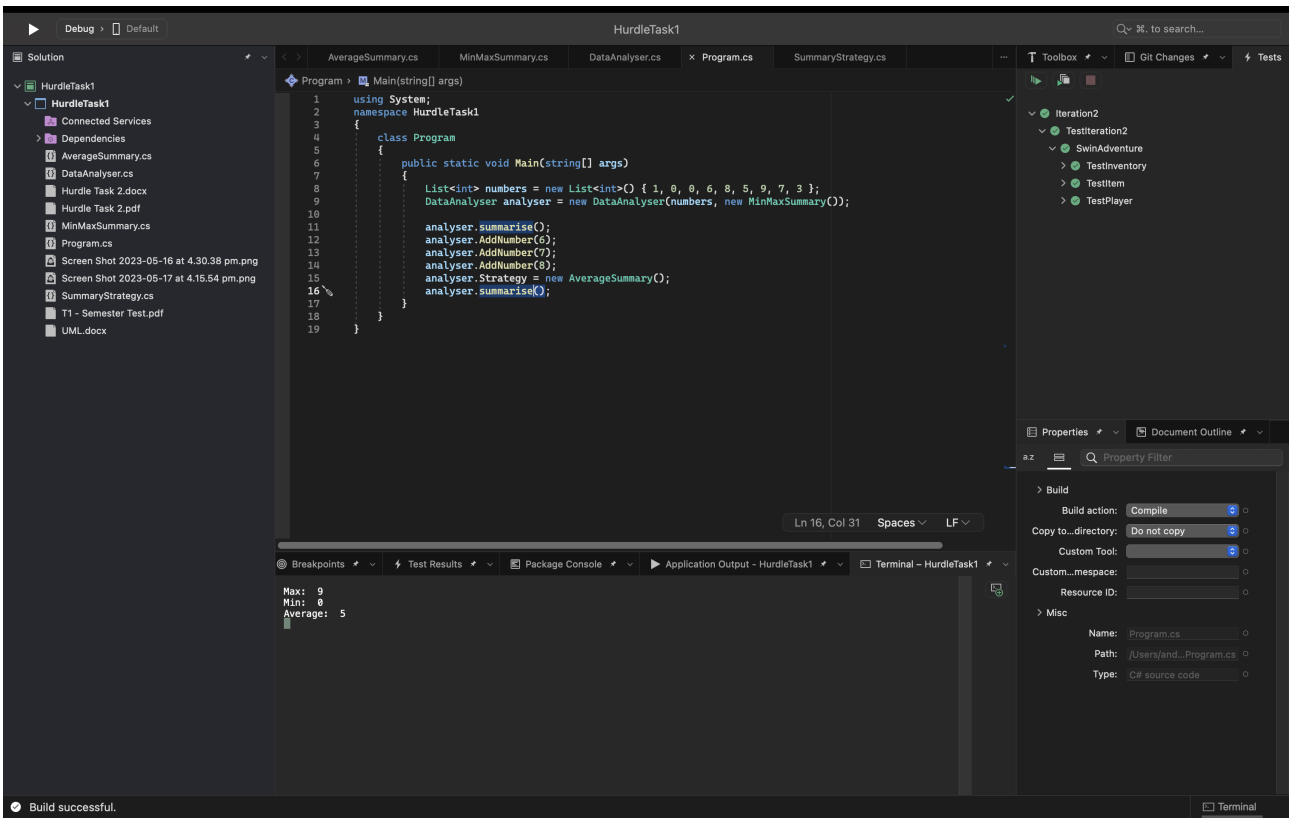
```
using System;
namespace HurdleTask1
{
    class Program
    {
        public static void Main(string[] args)
        {
            List<int> numbers = new List<int>() { 1, 0, 0, 6, 8, 5, 9, 7, 3 };
            DataAnalyser analyser = new DataAnalyser(numbers, new MinMaxSummary());

            analyser.summarise();
            analyser.AddNumber(6);
            analyser.AddNumber(7);
            analyser.AddNumber(8);
            analyser.Strategy = new AverageSummary();
            analyser.summarise();
        }
    }
}
```

```
using System;
namespace HurdleTask1
{
    public class DataAnalyser
    {
        private SummaryStrategy _strategy;
        private List<int> _numbers = new List<int>();



        public DataAnalyser() : this(new List<int>(), new AverageSummary())
        {

        }

        public DataAnalyser(List<int> numbers, SummaryStrategy strat)
        {
            _numbers = numbers;
            _strategy = strat;
        }


        public SummaryStrategy Strategy
        {
            get { return _strategy; }
            set { _strategy = value; }
        }

        public void AddNumber(int num)
        {
            _numbers.Add(num);
        }

        public void summarise()
        {
            _strategy.PrintSummary(_numbers);
        }

    }
}

```

```
using System;

namespace HurdleTask1
{
    public abstract class SummaryStrategy
    {
        public abstract void PrintSummary(List<int> numbers);

    }
}

```

```
using System;
namespace HurdleTask1
{
    public class MinMaxSummary : SummaryStrategy
    {

        private int Minimum(List<int> numbers)
        {
            int minimum = numbers[0];

            foreach (int num in numbers)
            {
                if (num < minimum)
                {
                    minimum = num;
                }
            }

            return minimum;
        }


        private int Maximum(List<int> numbers)
        {
            int maximum = numbers[0];

            foreach (int num in numbers)
            {
                if (num > maximum)
                {
                    maximum = num;
                }
            }

            return maximum;
        }


        public override void PrintSummary(List<int> numbers)
        {
            Console.WriteLine("Max:  " + Maximum(numbers));
            Console.WriteLine("Min:  " + Minimum(numbers));
        }

    }
}

```

```
using System;
namespace HurdleTask1
{
    public class AverageSummary: SummaryStrategy
    {

        private float Average(List<int> numbers)
        {
            int sum = 0;


            foreach (int num in numbers)
            {
                sum += num;
            }

            float average;


            average = (float) sum / numbers.Count;

            return average;
        }

        public override void PrintSummary(List<int> numbers)
        {

            Console.WriteLine("Average:  " + Average(numbers));
        }

    }
}

```


Debug
Default
HurdleTask1
Solution
HurdleTask1
HurdleTask1
Connected Services
Dependencies
AverageSummary.cs
DataAnalyser.cs
Hurdle Task 2.docx
Hurdle Task 2.pdf
MinMaxSummary.cs
Program.cs
Screen Shot 2023-05-16 at 4.30.38 pm.png
Screen Shot 2023-05-17 at 4.15.54 pm.png
SummaryStrategy.cs
T1 - Semester Test.pdf
UML.docx
AverageSummary.cs
MinMaxSummary.cs
DataAnalyser.cs
Program.cs
SummaryStrategy.cs
Program › Main(string[] args)
using System;
namespace HurdleTask1
{
class Program
{
public static void Main(string[] args)
{
List<int> numbers = new List<int>() { 1, 0, 0, 6, 8, 5, 9, 7, 3 };
DataAnalyser analyser = new DataAnalyser(numbers, new MinMaxSummary());
analyser.summarise();
analyser.AddNumber(6);
analyser.AddNumber(7);
analyser.AddNumber(8);
analyser.Strategy = new AverageSummary();
analyser.summarise();
}
}
}
Ln 16, Col 31
Spaces
LF
Breakpoints
Test Results
Package Console
Application Output - HurdleTask1
Terminal – HurdleTask1
Max: 9
Min: 0
Average: 5
Toolbox
Git Changes
Tests
Iteration2
TestIteration2
SwinAdventure
TestInventory
TestItem
TestPlayer
Properties
Document Outline
Property Filter
Build
Build action:
Compile
Copy to...directory:
Do not copy
Custom Tool:
Custom...mespace:
Resource ID:
Misc
Name:
Program.cs
Path:
/Users/and...Program.cs
Type:
C# source code
Build successful.
Terminal

Hurdle Task 2

1. **Describe the principles of polymorphism and how it was used in task 1.**

   The term Polymorphism means many forms. It is the ability for objects to present the same method but in different form according to the object's requirements. This provides extra flexibility in an object-oriented design. There are various types of polymorphism such as Method Overriding, overloading and the use of abstract classes.

   In task 1, the original design included a summarise method in DataAnalyser which took in a string parameter related to the summary methods in MinMaxSummary and AverageSummary which means the design consists of two summary methods. Suppose we are to include another summary class to provide a different type of summary, we would have to implement a new summary method. This can go on for N number of summary classes and methods, which is not very efficient.

   The new design provides an abstract class SummaryStrategy with an abstract method PrintSummary() that includes a list as the parameter. PrintSummary() is extended to both AverageSummary and MinMaxsSummary in which both methods override the abstract method to provide their own functionality. This allows for access to MinMaxSummary and AverageSummary through SummaryStrategy and makes it a lot easier since this gives us direct access to PrintSummary and which is linked to both MinMaxSummary and AverageSummary.

2. **Using an example, explain the principle of Abstraction.**

   The principle of abstraction allows you to represent real world objects by identifying the objects details and behaviours. When you identify key features and methods that represent the object, you have the tools to create a model that simulates those defined details and behaviours.

   Let's look at the clock example from this unit, abstraction in the clock's domain is the idea that the clock isn't made up of hours, seconds and minutes, the clock only needs to tick and reset for it be a clock. At a later stage the clock's detail that are specific to that clock can be added. That maybe be, what colour it is, the shape of the clock, whether its digital or quartz (maybe an atomic clock) but fundamentally all these clocks have the same characteristics and that is that they need to tick and provide the time in seconds, minutes, and hours and due to the nature of earth's spin, it needs to reset.

   The ways we achieve abstraction is through abstract classes and interfaces. An abstract class cannot be instantiated and acts as a blueprint for derived classes. It does not contain any concrete methods; these are defined when the class is extended, the abstract class doesn't tell us how a something should be implemented, it just tells us what needs to be done, the extension of the abstract class is responsible for the implementation. This allows us to create modular and reusable code by separating the interface (abstract class and interface) and the implementation of which the derived classes are responsible for which results in easier maintenance and extensibility as changes can be made in the derived classes without affecting the behaviour of the abstract class or interface.

3. **What was the issue with the original design in Task 1? Consider what would happen if we had 50 different summary approaches to choose from instead of just 2.**

   The original design still works and provides the desired output; however, the issues arise for when more summary approaches are added. If we have 50 different summary approaches, then the code will have 50 different summary methods that require its own code and need to be accessed through 50 different variables. This can still work but its

unnecessarily complex and inefficient design that can slow down the processing and incur a lot of costs.