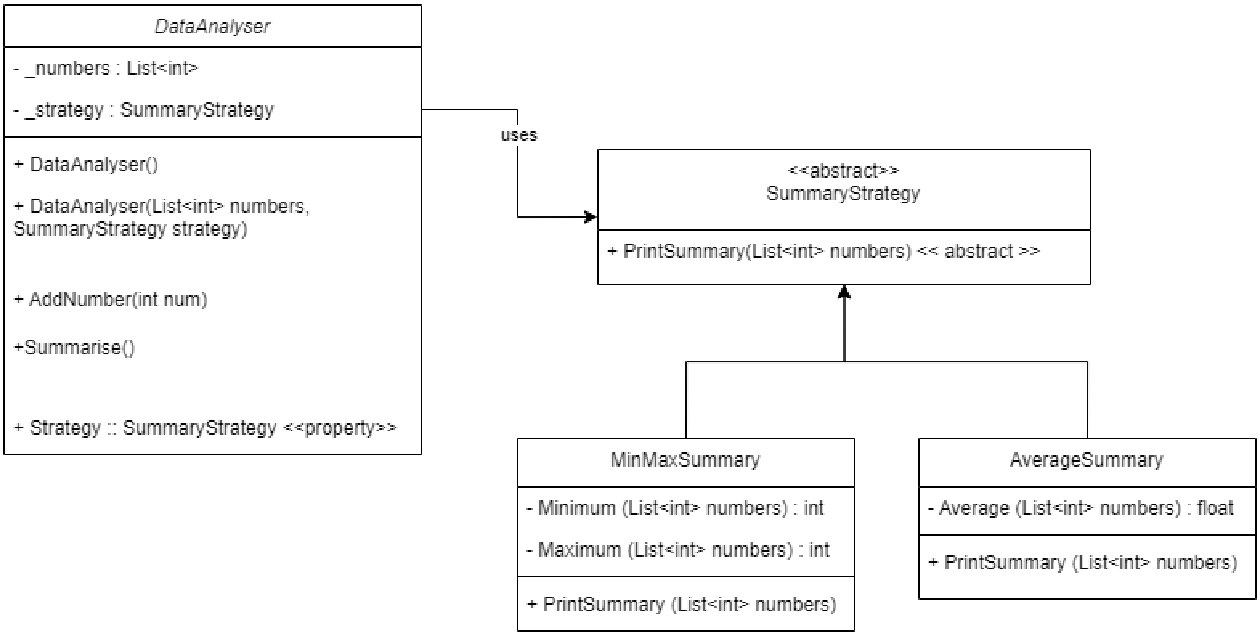
Swinburne University of Technology

COS20007 Object Oriented Programming

T1 - Semester Test

PDF generated at 22:27 on Monday 15th May, 2023

Produced by OOP Submission Formatter


DataAnalyser
- _numbers : List<int>
- _strategy : SummaryStrategy
+ DataAnalyser()
+ DataAnalyser(List<int> numbers, SummaryStrategy strategy)
+ AddNumber(int num)
+Summarise()
+ Strategy :: SummaryStrategy <<property>>
uses
<<abstract>>
SummaryStrategy
+ PrintSummary(List<int> numbers) << abstract >>
MinMaxSummary
- Minimum (List<int> numbers) : int
- Maximum (List<int> numbers) : int
+ PrintSummary (List<int> numbers)
AverageSummary
- Average (List<int> numbers) : float
+ PrintSummary (List<int> numbers)

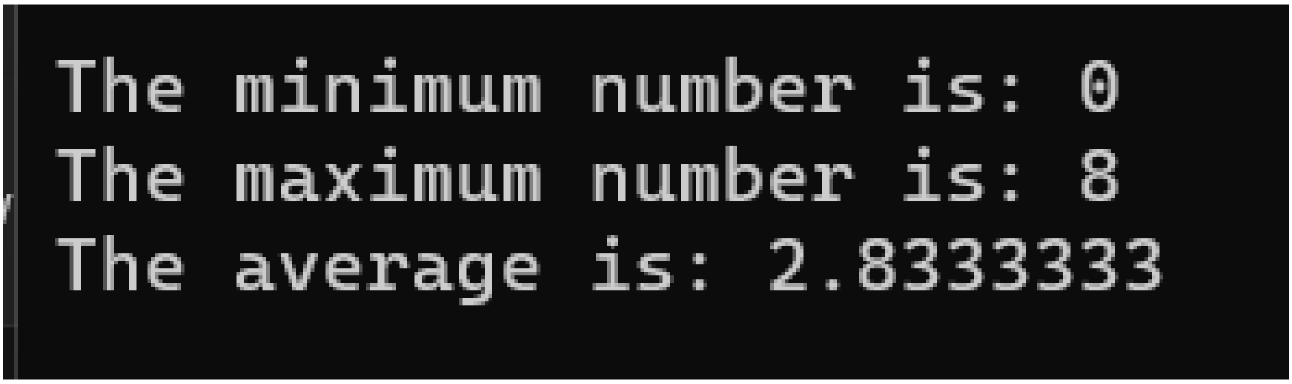
```
using System;

namespace T_
{
    public class Program
    {
        public static void Main()
        {

            List<int> list = new List<int> {1,0,3,8,3,0,6,8,2 };

            MinMaxSummary minmaxSummary = new MinMaxSummary();
            AverageSummary averageSummary = new AverageSummary();

            DataAnalyser data = new DataAnalyser(list, minmaxSummary);

            data.Summarise();

            data.AddNumber(0);
            data.AddNumber(1);
            data.AddNumber(2);

            data.Strategy = averageSummary;

            data.Summarise();
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace T_
{
    public class DataAnalyser
    {
        //private local variables
        private List<int> _numbers;
        private SummaryStrategy _strategy;

        //default constructor with average summary as default strategy
        public DataAnalyser() : this (new List<int>(), new AverageSummary()) { }

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

        //add a number method
        public void AddNumber(int num)
        {
            _numbers.Add(num);
        }

        //summarize method to print summary
        public void Summarise()
        {
            _strategy.PrintSummary(_numbers);
        }

        //property for _strategy, read and write
        public SummaryStrategy Strategy
        {
            get
            {
                return _strategy;
            }
            set
            {
                _strategy = value;
            }
        }
    }

}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace T_
{
    public class MinMaxSummary : SummaryStrategy
    {

        //minimum and maximum variables that identify the mimimum and maximum values in a list
        private int Minimum (List<int> numbers)
        {
            int min = numbers[0];
            foreach (int num in numbers)
            {
                if (num < min)
                {
                    min = num;
                }
            }
            return min;
        }

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

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

        //method to print the minimum and maximum number of a list using the variables above
        public override void PrintSummary(List<int> numbers)
        {
            Console.WriteLine($"The minimum number is: {Minimum(numbers)}");
            Console.WriteLine($"The maximum number is: {Maximum(numbers)}");
        }

    }
}
```

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace T_
{
    public class AverageSummary : SummaryStrategy
    {
        //average variable that calculates the average of all the numbers a the list
        private float Average (List<int> numbers)
        {
            float sum = 0;
            float average;

            foreach (int num in numbers)
            {
                sum += num;
            }
            average = sum / numbers.Count;

            return average;
        }

        //method to print the average number of a list using the variable above
        public override void PrintSummary(List<int> numbers)
        {

            Console.WriteLine($"The average is: {Average(numbers)}");
        }
    }
}
```



```
The minimum number is: 0
The maximum number is: 8
The average is: 2.8333333
```

1. Polymorphism is principle In OOP that allows objects to have different forms, which in turn allows these objects to be used in different ways depending on the context. The 2 ways polymorphism can be achieved in OOP are overloading and overriding. Overloading allows a method to have multiple ways of being called, by accepting different parameters. Overriding allows different implementations to a method by changing them in a subclass. In task 1, we used polymorphism in both the ways. The DataAnalyser class had two different constructor methods accepting different parameters, this method is now overloaded. Further, the subclasses of the abstract class SummaryStrategy overrided the PrintSummary method to have a more specific function, allowing the same method to have different forms.

2. Abstraction is a design process that involves identifying the essential entities, their roles, responsibilities, and relationships in the problem space. It allows simplified representations of a program by focusing on the core features, rather than the details and its implementation. This allows the system design to be flexible and reusable since it is not specific to one logic or language, so theoretically it could be implemented in different ways.

For example, in a space shooter game, some of the entities would be player ship, and enemy ship. The roles of the ships would be to move and shoot bullets, they way these roles are implemented (whether they will have their own class, or derive from one ship class, or any other possible implementation) is not outlined here so it is flexible, allowing the design to be abstract and not detailed/specific.

3. The problem with original design of Task 1 is that it was not scalable. The DataAnalyzer class had the summarize method hard coded to accept only two strategies, so if we had 50 different strategies instead of just the 2, we would have 50 different conditional statements, and would have to modify it each time we wanted to add a new one. This makes the code hard to read and maintain, in addition to being unscalable.