

OOP

by Sophie Novak

FILE	2769_SOPHIE_NOVAK_OOP_93264_1098175649.DOCX		
TIME SUBMITTED	23-MAR-2018 02:13PM (UTC+0000)	WORD COUNT	2931
SUBMISSION ID	85449890	CHARACTER COUNT	19485

UWTSD

Application Design and Development

Object Oriented Programming

Sophie Novak (P149298)
23/03/2018

Table of Contents

Abstract	2
Object Oriented Programming Principles	3
The difference between a class and an object	3
This represents the class	3
This represents the object	3
The important implications of inheritance and polymorphism in the development of large applications	4
Polymorphism	4
Inheritance	5
Interface or class	6
Advantages and disadvantages of Dependency-Inversion principle	7
UML Design	8
Testing	10
References	11
Appendix	13
Spoon	13
Circle	14
Shopping	15
Game	17

Abstract

This report includes object orientated principles, which within that contains what the difference is between an object and a class, the importance of polymorphism and inheritance within large applications, when a programmer should decide to choose an interface over a class and some pros and cons about the dependency inversion principle. This report also contains a UML diagram which was created with the code provided. Finally, it includes a definition of software testing and explanations about black box and white box testing.

Object Oriented Programming Principles

The difference between a class and an object

There are many differences between an object and a class, some of the main differences are that an object is an instance of a class [1]. This means within a class is an object and the object defines what the class can do. Furthermore, all objects belong to a class and every class contains one or more objects [2]. Additionally, classes are declared by using the keyword 'class' whereas objects are declared by using the keyword 'new' which carries the constructors of the class to initialise and create an object [3]. For example:

This represents the class

```
class Student {  
    private int _result;  
  
    public Student(int Result){  
        _result = Result;  
    }  
  
    public void SetResult(string value){  
        _result = value;  
    }  
    Public int GetResult(){  
        Return _result;  
    }  
}
```

This represents the object

```
Student result1 = new Student(70);
```

The important implications of inheritance and polymorphism in the development of large applications

Polymorphism

Polymorphism is when a child class replaces a parent class method with the child's own class.

Polymorphism is very important when using large applications, this is because it does not replicate data, so there are not thousands of unnecessary lines of code. There are three ways that polymorphism does not replicate code, the first technique is called: Virtual method pattern, this pattern contains the word 'virtual' whilst implementing the polymorphism. The word 'virtual' within the application means that the derived classes will be able to change the way the method is defined, furthermore, providing their own definition [4]. This particular method would be important to be aware of, but it is not the best method to use whilst using a large application. The reason for this is that the programmer can only use one child class to define the parent class, whereas in a larger application the parent class should be able to define more than one child class. An example of the how the virtual method pattern is implemented is shown below:

```
public class ParentClass
{
    public virtual type MethodName(argDeclaration1...)
{
}
}
```

Another method which could be used in large applications is the override method pattern, this pattern does not include the word virtual, it contains the word 'override' instead whilst carrying out the polymorphism technique. 'Override' within the pattern will override the child class already defined by the parent class and create a new and defined child class. The override method is used particularly when creating large applications, this is because the programmer can use the same parent class for how many child classes needed, this will reduce repetition of code. Overall, making a more efficient website. An example how the override method pattern is applied is shown below:

```
public class ChildClass: ParentClass
{
    public override type MethodName(argDeclaration1...)
{
}
}
```

The final way to implement polymorphism is by using the abstract method pattern this uses the word 'abstract' within the code. The abstract method pattern is used to prevent an object being

directly created from the parent class and when the programmer wishes not to provide code for the method within the parent class as the child class will implement it. This method is again, particularly good for large applications, the reason for this is that the amount of code is cut down yet again. The child class will be carrying all the code, so the programmer will not have to repeat code at all, also it will be less demanding for the programmer to use this method. This is because the programmer will not have to remember what is in the child class whilst implementing it into the parent class because the child class holds all the information, not the parent class. Overall, this method is the best for large applications, an example of the abstract method pattern is implemented is shown below:

```
abstract public class ParentClass
{
    abstract public type MethodName(argDeclaration1...);
}
```

Inheritance

Inheritance is fundamental within object-oriented programming, inheritance allows the programmer to define a child class which reuses, also known as inherits, or modifies the behaviour of a parent class. The classes whose components are inherited is known as the 'base class' and the classes which inherit the components of the base class is known as the 'derived class' [5]. However, not all the components of the base class are inherited by the derived classes, such as:

Static constructors, which initialize static data within a class.

Instance constructors, which is called when a new instance of a class needs to be created.

Finalizers, which are called to destroy the class instances.

All other components of the base class are inherited by the derived classes but depending on their accessibility results in whether they are visible or not [5]. Furthermore, there are four types of accessibility, which are: Private, protected, internal and public.

Private components are only visible in the derived class which is nested in the base class, otherwise they will not be visible in the derived class alone.

Protected components are only visible in derived classes.

Internal components are only visible in derived classes which are located within the same connection as the base class, not if they were located elsewhere.

Public components are visible in the derived classes and are a section of the derived class' public interface.

Inheritance is important in the implication of large applications because the child class is able to reuse or modify the behaviour of a parent class, this does not require much code as the parent class will hold majority of the code and the child class will just implement it.

Interface or class

An interface must be fulfilled by a class, for example, it defines the capabilities that is available in the class. A class is a specification on how to provide a service. The reason why programmers will use an interface over a class is when the programmer is writing an API, the reason for this is that in an API, the code must be loosely coupled [6]. An interface should also be used when the type of data is different but the functionality which is requires is exactly the same. The interface requires no code as the class must provide it and the class will implement the interface providing methods, so this will require less work that it would to create a whole new class. Also, interfaces can inherit from just one parent class, but it can implement from multiple interfaces, so the application can hold many different interfaces within one class. Overall, interfaces are easier to implement than classes and require less code, and the programmer would use an interface instead of a class when they have many different interfaces as a class can hold many interfaces whereas a class can only hold one other class.

Advantages and disadvantages of Dependency-Inversion principle

Dependency principles typically include parent classes which are designed to perform a task and use the child classes to provide the functionality which will support the task in which is being carried out. This, in turn, makes the parent classes depend on the child classes.

Inversion is where the parent class defines the interface that the child classes must implement. The dependency is inverted as the child classes depend on the interface which is provided by the parent class.

An advantage of using the dependency inversion principle is that it will lead to more flexible code as the changes which are involved in the child class will not require the parent class to change. Moreover, it is more efficient to build higher level applications without thinking about the implementation details which are included in the subtasks. Dependency inversion principle achieves loosely coupling as each component has little or no knowledge of the definitions of the other components. Furthermore, this achieves testability and replaceability as components within a loosely coupled system are able to be replaced with other implementations which provide similar services [7].

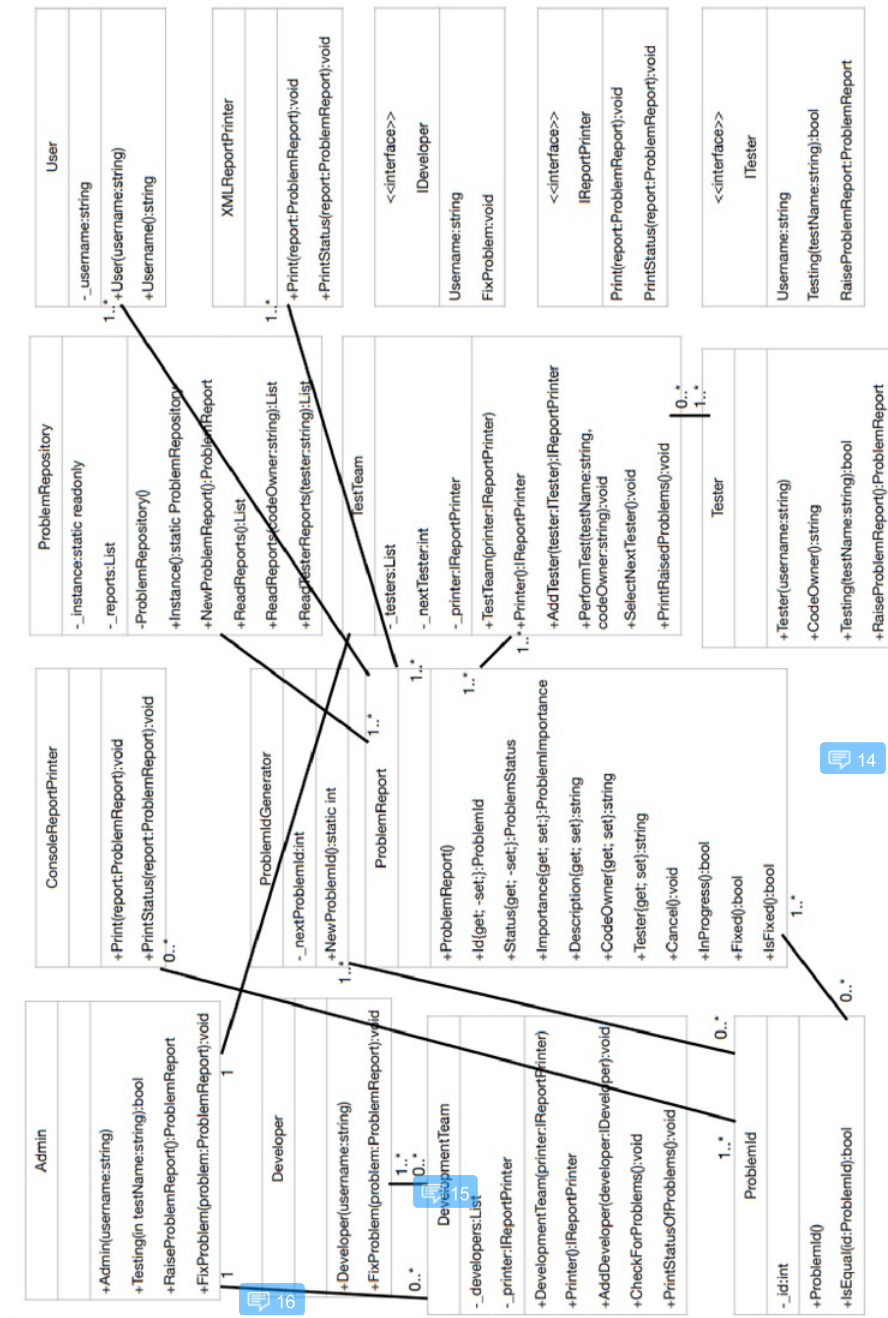
 12

Some disadvantages of using the dependency inversion principle is that the application could potentially become tightly coupled which means: testing the application will become difficult, as well as, the parallel development [8]. Another disadvantage is that the programmer needs a dependency inversion container which needs to be configured. This container has required the capabilities to inject into the required places within the services [7].

 13

UML Design

Below is a UML Design for the code provided for the assignment.



Within the above UML diagram, there are all the classes which were created for the application, including the Admin class, the Problem Report class, the Tester class etc. Within each of them tables include the attributes of each class, if the class did have attributes, the constructors, methods and properties, if the classes provided them. Most of the tables connect to each other, with majority connecting via the 'one to many' technique. For example; the Developer may have 0 or more Developer team whereas the Developer team will have more than one developer within it.

There are some design principles included within the classes, the most obvious one was the ProblemRepository class had the Singleton pattern within, as the keyword 'static' was applied. This was because some of the items within the class are private, to use them the public static property must be added to gain access to it.

Testing

Testing the software that has been created is essential to have a flowing application. Software testing includes a process of finding software bugs, validating and verifying that the code works within the application [9]. Software testing helps to discover errors or missing requirements, software testing can either be done manually or using tools, such as white box and black box testing [10].

Black box testing is a method within software testing which the internal structure or design of the item being tested on is not known to the software tester [11]. In cases where the code of the software is unavailable, the test cases which are generated by the black box testing technique require that the quality of testing remains the same [12]. Black box testing does not have any knowledge of the internal system working and only examines the necessary aspects of the system [13]. Black box testing is efficient for large amounts of code, but only a selected amount of test scenarios are executed [13].

White box testing is a method within software testing which the internal structure or design of the item being tested is known to the tester [11]. White box testing is used mostly for verification [14], to verify to the programmer that everything works perfectly, and nothing has been left out of the requirements. White box testing is used for three types of testing: Unit testing, which tests individual software or hardware groups of units which are related [14]. Integrating testing, which tests the software components or the hardware components or both when they are combined. This is to see the connection between the hardware and software [14]. Finally, regression testing, which includes retesting the components selectively to verify that there are no errors [14].

References

- [1] "Java point," [Online]. Available: <https://www.javatpoint.com/difference-between-object-and-class>. [Accessed 07 March 2018].
- [2] Mithun, "Quora," 09 May 2017. [Online]. Available: <https://www.quora.com/What-is-the-difference-between-class-and-object-in-OOP>. [Accessed 07 March 2018].
- [3] "Javarevisited," [Online]. Available: <http://javarevisited.blogspot.co.uk/2012/12/difference-between-class-and-object-in-oops-java.html>. [Accessed 07 March 2018].
- [4] "RB Whitaker's Wiki," [Online]. Available: <http://rbwhitaker.wikidot.com/c-sharp-abstract-classes>. [Accessed 20 March 2018].
- [5] "Inheritance in C# and .NET," Microsoft, 16 August 2017. [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/inheritance>. [Accessed 20 March 2018].
- [6] J. James, "TechRepublic," 18 July 2010. [Online]. Available: <https://www.techrepublic.com/blog/software-engineer/when-to-use-interfaces-instead-of-classes/>. [Accessed 20 March 2018].
- [7] F. Despoudis, "codeburst.io," 18 July 2017. [Online]. Available: <https://codeburst.io/understanding-solid-principles-dependency-injection-d570c15560ab>. [Accessed 20 March 2018].
- [8] P. Rana, "C# Corner," 07 July 2015. [Online]. Available: <https://www.c-sharpcorner.com/UploadFile/pranayamr/dependency-inversion-principle/>. [Accessed 20 March 2018].
- [9] "ISTQ EXAM CERTIFICATION," [Online]. Available: <http://istqbexamcertification.com/what-is-software-testing/>. [Accessed 20 March 2018].
- [10] "GURU99," [Online]. Available: <https://www.guru99.com/software-testing-introduction-importance.html>. [Accessed 20 March 2018].
- [11] "Software Testing Fundamentals," [Online]. Available: <http://softwaretestingfundamentals.com/differences-between-black-box-testing-and-white-box-testing/>. [Accessed 20 March 2018].
- [12] E. K. S. Harsh Bhasin, "Black Box Testing based on Requirement Analysis and Design Specifications," *International Journal of Computer Applications (0975 – 8887)*, vol. 87, no. 18, pp. 36-40, 2014.
- [13] F. K. Ehmer Kahn, "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques," *(IJACSA) International Journal of Advanced Computer Science and Applications*, vol. 3, no. 6, pp. 12-15, 2012.

- [14] N. Gupta, "Different Approaches to White Box Testing to Find Bug," *International Journal of Advanced Research in Computer Science & Technology (IJARCST 2014)*, vol. 2, no. 3, pp. 46-49, 2014.

Appendix

Spoon

Spoon.cs

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

namespace _1Spoons
{
    class Spoon
    {
        //Attributes
        private double _total;

        //Constructors
        public Spoon(int initialCount)
        {
            _total = initialCount;
        }

        //Properties
        public double GetTotal()
        {
            return _total;
        }

        public void SetTotal(int value)
        {
            _total = value;
        }

        //Methods
        public void AddMoreSpoons(int spoonCount)
        {
            _total = _total + spoonCount;
        }
    }
}
```

Program.cs

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

namespace _1Spoons
{
    class Program
    {
        static void Main(string[] args)
        {
            int initialSpoonCount = 0;
            int extraSpoon = 1;
            double totalSpoons;
        }
    }
}
```

```

        Spoon spoon = new Spoon(initialSpoonCount);
        spoon.AddMoreSpoons(extraSpoon);
        totalSpoons = spoon.GetTotal();
        Console.WriteLine("The total is {0} spoon(s) and the price is {1}",
totalSpoons, totalSpoons * 0.20);

        extraSpoon = 1;
        spoon.AddMoreSpoons(extraSpoon);
        totalSpoons = spoon.GetTotal();
        Console.WriteLine("The total is {0} spoon(s) and the price is {1}",
totalSpoons, totalSpoons * 0.20);

        Console.ReadKey();
    }
}

```

Circle

Overlap.cs

```

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

namespace Circle
{
    class Overlap
    {
        //Attributes
        private int _x1;
        private int _y1;
        private int _radius = 15;
        private int _x2;
        private int _y2;
        private int _radius2 = 10;
        private bool overlapTest;
        private int _dist;

        //Constructors

        //Properties

        //Methods
        public void Distance()
        {
            ((_x1 - _x2) * (_x1 - _x2) + (_y1 - _y2) * (_y1 - _y2)) < (_dist * _dist)
        }
    }
}

```

Program.cs

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

namespace Circle
{
    class Program
    {
        static void Main(string[] args)
        {
            Overlap circle = new Overlap(bool);
        }
    }
}
```

Shopping

Basket.cs

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

namespace Shopping
{
    class Basket : ICD, IDVD, IBook
    {
        //Attributes
        private ITax _tax;

        //Constructors

        //Properties

        //Methods
        public void CDTax()
        {
            Console.WriteLine("The price of the CD without tax is £1.50 and the price including tax is {0}", 1.50 - (1.50 / 100) * 10 );
        }

        public void DVDTax()
        {
            Console.WriteLine("The price of the DVD without tax is £2.70 and the price including tax is {0}", 2.50 - (2.50 / 100) * 12);
        }

        public void BookTax()
        {
            Console.WriteLine("The price of the Book without tax is £3.00 and the price including tax is {0}", 3.00 - (3.00 / 100) * 0);
        }
    }
}
```

IBook.cs

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

namespace Shopping
{
    interface IBook
    {
        void BookTax();
    }
}
```

ICD.cs

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

namespace Shopping
{
    interface ICD
    {
        void CDTax();
    }
}
```

IDVD.cs

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

namespace Shopping
{
    interface IDVD
    {
        void DVDTax();
    }
}
```

ITax.cs

```
using System;
using System.Collections.Generic;
```

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

```
namespace Shopping
{
    interface ITax
    {
        void Tax();
    }
}
```

Program.cs

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

namespace Shopping
{
    class Program
    {
        static void Main(string[] args)
        {
            ICD cd;
            IDVD dvd;
            IBook book;

            cd = new Basket();
            cd.CDTax();

            dvd = new Basket();
            dvd.DVDTax();

            book = new Basket();
            book.BookTax();

            Console.ReadKey();
        }
    }
}
```

Game

PlayerOne.cs

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

namespace Game
{
    class PlayerOne
    {
        //Attributes
    }
}
```

```

        private string _letter;
        private string number1;

        //Constructors
        public PlayerOne(string letter)
        {
            _letter = letter;
        }

        public PlayerOne(string letter, string number1) : this(letter)
        {
            this.number1 = number1;
        }

        //Properties
        public void setLetter(string value)
        {
            _letter = value;
        }

        public string getLetter()
        {
            return _letter;
        }
        //Methods
    }
}

```

PlayerTwo.cs

```

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

namespace Game
{
    class PlayerTwo
    {
        //Attributes
        private string _letter;
        private string number2;

        //Constructors
        public PlayerTwo(string letter)
        {
            _letter = letter;
        }

        public PlayerTwo(string letter, string number2) : this(letter)
        {
            this.number2 = number2;
        }

        //Properties
        public void setLetter(string value)

```

```

        {
            _letter = value;
        }

        public string getLetter()
        {
            return _letter;
        }
        //Methods
    }
}

```

Program.cs

```

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

namespace Game
{
    class Program
    {
        static void Main(string[] args)
        {
            string number1;
            string number2;

            Console.WriteLine("What letter do you choose?");
            number1 = Console.ReadLine();
            PlayerOne player1 = new PlayerOne("{0}", number1);

            Console.WriteLine("What letter do you choose?");
            number2 = Console.ReadLine();
            PlayerTwo player2 = new PlayerTwo("{0}", number2);

            Console.WriteLine("Player one chooses {0} and player two chooses {1}",
number1, number2);

            Console.ReadLine();
        }
    }
}

```

FINAL GRADE

45/100

GENERAL COMMENTS

Instructor

Knowledge OOPs concepts (30) 11

Discussion of design principles (10) 7

Knowledge of testing (10) 6

UML (20) : 13

Exercise 1 (5) 3

Exercise 2 (5) 3

Exercise 3 (10) 1

Exercise 4 (10) 1

Classes vs Objects: from the explanation provided it is clear that you don't know the difference, which is particularly concerning since this is fundamental to OOPs

Inheritance and Polymorphism: not totally convinced you understand these concepts or their importance.

Interfaces: don't think you understand interfaces and you have missed some important points

Dependency Inversion: quite a good explanation, again there are aspects of your explanation that are not very clear.

Software Testing: you have briefly covered white, black and unit testing. But you could have outlined more of the benefits of these approaches.

UML: the relationships shown in the class diagram

are a bit random, although you have the correct idea. Inheritance not shown. You should have shown sequence diagrams for common scenarios.

Ex1: You counted the number of spoons but didn't calculate the total cost

Ex2: You should have created a Circle class

Ex3: you needed different products that inherited from a Product class or an interface IProduct stored on a list in the Basket

Ex4: started exercise.

PAGE 1

PAGE 2

PAGE 3

PAGE 4



Comment 1

Confused and the "object defines what the class can do" is incorrect



Comment 2

explain



Comment 3

They can but they don't have to

PAGE 5



Comment 4

This is wrong, or you it might be the just the way you have expressed it.



Comment 5

??

PAGE 6



Comment 6

There is a vague understanding of the idea of abstraction, but it is just not clear enough to be sure that you understand it

PAGE 7



Comment 7

expand on this idea



Comment 8

classes



Comment 9

not sure what you are trying to say here



Comment 10

I am not convinced by this argument



Comment 11

The main purpose of interfaces is to defined "roles" that a class can play. The "role" should be defined by the class requiring the functionality rather than the classes implementing the interface.

Interfaces allow us to subdivide our code at a more granular level.

PAGE 8



Comment 12

nice point



Comment 13

explain

PAGE 9



Comment 14

A number of these relationships are incorrect, in part because you missed out the implementation of the interfaces



Comment 15

No, this a unidirectional relationship



Comment 16

No, you missed out the interfaces IDeveloper and ITester

PAGE 10



Comment 17

actually the developer has no knowledge of belonging to a development team, the team has a list of developers



Comment 18

others include SRP and ISP

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15



Comment 19

Should have used a Circle class

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20
