

PD_ADAD_ASS

by Peter Davies

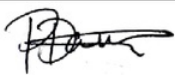
FILE	2780_PETER_DAVIES_PD_ADAD_ASS_93264_1992318988.PDF		
TIME SUBMITTED	23-MAR-2018 11:36PM (UTC+0000)	WORD COUNT	9353
SUBMISSION ID	85497522	CHARACTER COUNT	57110



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

School of Applied Computing

Student Name:	Peter Davies	Student P number:	P079868
Programme:	BSc Web Development		
Module:	M5X01025	Lecturer:	Mike Dacey
Assignment Title:	Application Design & Development		
Hand Out Date:	2017	Submission deadline:	23/03/18

Disclosure:	I declare that this assignment is all my own work and that I have acknowledged all materials used from the published or unpublished works of other people. All references have been duly cited.		
Student's Signature:		Date:	23/03/18

Contents

1. Introduction.....	1
2. Object Orientated Programming Principles.....	2
2.1 Classes & Objects.....	2
2.2 Inheritance & Polymorphism.....	5
2.3 Interfaces.....	11
2.4 Benefits & Drawbacks of Dependency-Inversion.....	17
3. Software Testing.....	19
3.1 Black Box Testing.....	19
3.2 White Box Testing.....	19
3.3 Unit Testing.....	20
4. Object Orientated Programming.....	20
4.1 Exercise One.....	20
4.1.1 Exercise One Testing.....	20
4.2 Exercise Two.....	25
4.2.1 Exercise Two Testing.....	28
4.3 Exercise Three.....	31
4.3.1 Exercise Three Testing.....	35
5. UML Diagrams.....	37
5.1 Class Diagrams.....	38
6. Critical Reflection.....	39
7. References.....	40
8. Bibliography.....	43
9. Appendices.....	45

Introduction

The following report documents the assignment for module Application Design & Development. To begin, research is done into the fundamental aspects of object orientated programming, these include; classes & objects, inheritance & polymorphism, interfaces and the dependency inversion principle.

Section two puts theoretical principles into practice with attempts at a range of programming exercises using an object orientated approach. Testing after research into common testing practices is performed and programs improved. Finally, in section three, source code has been provided and a class diagram conceived from analysing the code.

A reference list and bibliography complete the report citing all works used in this report and all material used to aid the completion of this assignment.

Object Orientated Programming Principles

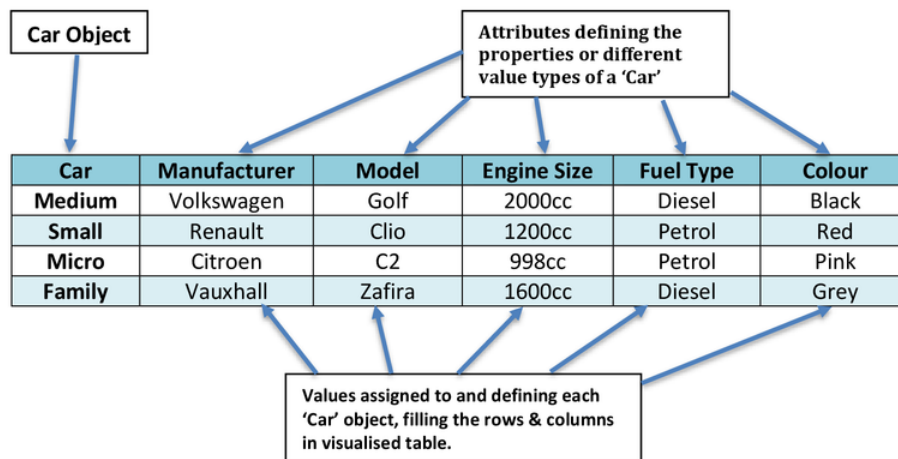
2.1 Classes & Objects

In object orientated programming, a class defines the structure for defining objects, it defines the names and types of variables that can exist in an object. It can be thought of as a container which holds data and code, which performs a particular job [1] or a template for defining objects where it can be thought of as a type, with objects being variables of that type [2].

An object is just that, an 'object', 'thing' or representations of entities in the real world, of which attributes can be used to describe that object [3], for example; a car is an object and has attributes or metadata which can be attached to it. A car will have a manufacturer, model name, engine size, fuel type, colour etc. All of these attributes will be defined in the class, and their values set when creating an object, unless they have been defined within the class.

To visualise a class and an object it is understood that a class can be thought of as a table within a database where the database is the overall program, and the class is a table within that program. Within the table, columns are used to separate different data types, these can be thought of as the attributes within a class. Finally, objects are made up of the data which fills the table rows & columns.

Utilising the car example, a table is outlined below to aid visualisation of classes and objects.



The above example can be further understood by implementing into code to produce a small program made up of a car class, and car objects; Medium, Small, Micro & Family.

First the class 'Car' is created, and attributes defined. *Note the naming convention for the attributes begins with an underscore, this is to distinguish the attributes from conventional variables.

Attribute declaration used to define properties of different cars. The structure for defining objects which will become instances of this class.

```
class Car
{
    // Attributes

    private string _manufacturer;
    private string _model;
    private int _engineSize;
    private string _fuelType;
    private string _colour;
}
```

Properties for each attribute are then 'get' and 'set' using the following code. This allows values for each object created to be passed in and pulled out in the main program file.

```
/* SET VALUES */
4 references
public void SetManufacturer(string value)
{
    _manufacturer = value;
}
4 references
public void SetModel(string value)
{
    _model = value;
}
4 references
public void SetEngineSize(int value)
{
    _engineSize = value;
}
4 references
public void SetFuelType(string value)
{
    _fuelType = value;
}
4 references
public void SetColour(string value)
{
    _colour = value;
}
```

```
/* GET VALUES */
4 references
public string GetManufacturer()
{
    return _manufacturer;
}
4 references
public string GetModel()
{
    return _model;
}
4 references
public int GetEngineSize()
{
    return _engineSize;
}
4 references
public string GetFuelType()
{
    return _fuelType;
}
4 references
public string GetColour()
{
    return _colour;
}
```

It is then possible to create each 'Car' object within the program and assign values to each of the attribute fields within the class.

Create new
'Car' Object

```
/* MEDIUM SIZE CAR OBJECT */
Car Medium = new Car();
Medium.SetManufacturer("Volkswagen");
Medium.SetModel("Golf");
Medium.SetEngineSize(2000);
Medium.SetFuelType("Diesel");
Medium.SetColour("Black");
```

Set values
to each
attribute
within the
'Car' class.

The above screen shot represents the creation of the 'Medium' Car object instance of the car class. This process is repeated for each instance, then printed out in the console to demonstrate the visualisation of a database table.

The results seen below show how each 'Car' object has been created with different values assigned to each different object, all using the same class.

MANUFACTURER	MODEL	ENGINE SIZE	FUEL TYPE	COLOUR
Volkswagen	Golf	2000	Diesel	Black
Renault	Clio	1200	Petrol	Red
Citroen	C2	998	Petrol	Pink
Vauxhall	Zafira	1600	Diesel	Grey

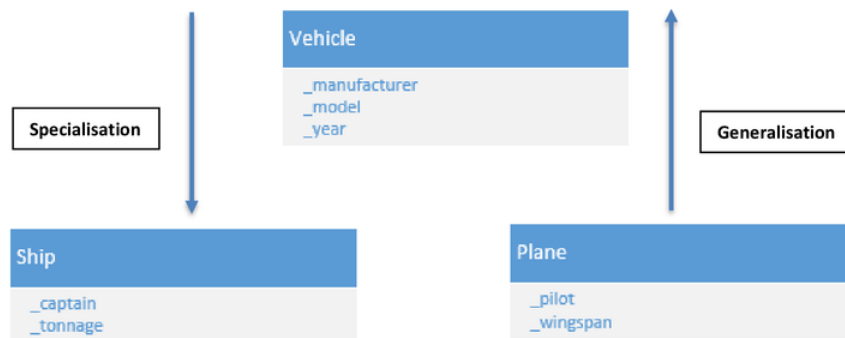
This is evidence that object orientated programming has a number of benefits. It aids programmers in solving larger problems, the above code could be modified and reused in other programs easily. It also shows that once an object is created, knowledge of its implementation is not necessary for its use, also known as encapsulation. Programmers also have the ability to hide certain parts of an object using the 'private' and 'public' keywords, another form of encapsulation. Older programs which need to be updated or modified can be done so easier due to the object orientated program being broken down in to smaller, more manageable sections of code [4].

2.2 Inheritance & Polymorphism

Inheritance is similar to that of database normalisation where repetition of fields is avoided in different tables. For example, if two classes are created; a ship class and a plane class, it can be said that both the ship class and plane class will share general properties; a manufacturer, a model, a year of manufacture etc. as well as many more specialised attributes for each such as; a captain or tonnage for a ship, or a wingspan, or pilot for a plane.

Inheritance allows programmers to create a parent class, aka, a 'super class' or 'base class' which contains attributes and methods which are shared commonly between children classes aka 'sub classes' or 'derived class'. In relation to the aforementioned example, this allows classes such as 'ship' class or 'plane' class to inherit general attributes and properties from a parent 'vehicle' class and therefore allowing programmers to organise and reuse code more efficiently [5], [6].

This can be best seen using UML diagrams. Below shows the parent 'Vehicle' class containing the general properties of which every vehicle will share. The children classes 'Ship' and 'Plane' then contain more properties specific to that vehicle type.



Derived classes 'Ship' & 'Plane' could be further split up into types of 'Ship' or 'Plane'. For example, Ferry, Cargo, Oil Tanker are all types of ships, each with their own specific properties, whilst still having properties inherited from the existing classes. This shows how inheritance allows a programmer to break down relationships to be able to organise & reuse code, especially useful in the case of implementing large programs.

In the example code below, it can be seen that the parent class 'Vehicle' contains two methods, 'PrintManufacturer()' and 'PrintYear()', each printing out a message to the console when a 'Vehicle' object is created and the methods called in the main program.

```

class Vehicle
{
    // Attributes
    // Constructors
    // Properties
    // Methods
    2 references
    public void PrintManufacturer()
    {
        Console.WriteLine("Who made me?");
    }
    2 references
    public void PrintYear()
    {
        Console.WriteLine("When was I made?");
    }
}

```

Methods

```

Console.WriteLine("VEHICLE CLASS");
Vehicle vehicle = new Vehicle();
vehicle.PrintManufacturer();
vehicle.PrintYear();

```

Object Creation

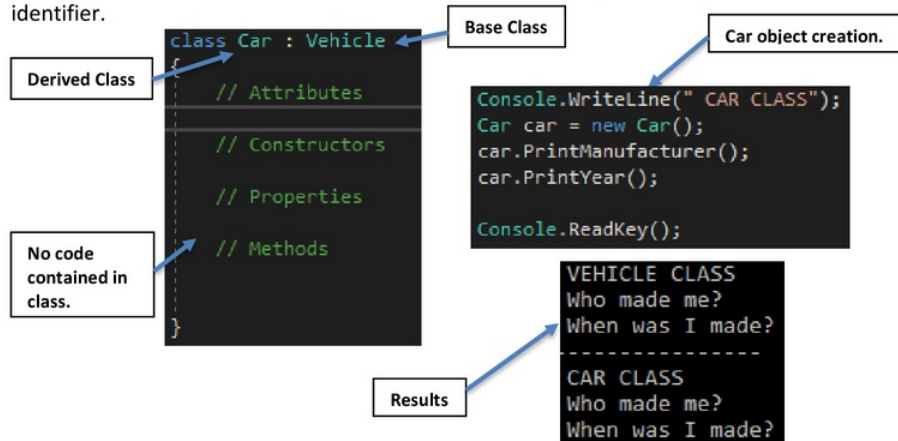
Results

```

VEHICLE CLASS
Who made me?
When was I made?

```

An empty child class 'Car' is then created which will inherit these methods from the 'Vehicle' class. To inherit from the base class 'Vehicle' a semi colon is placed after the derived classes' identifier.

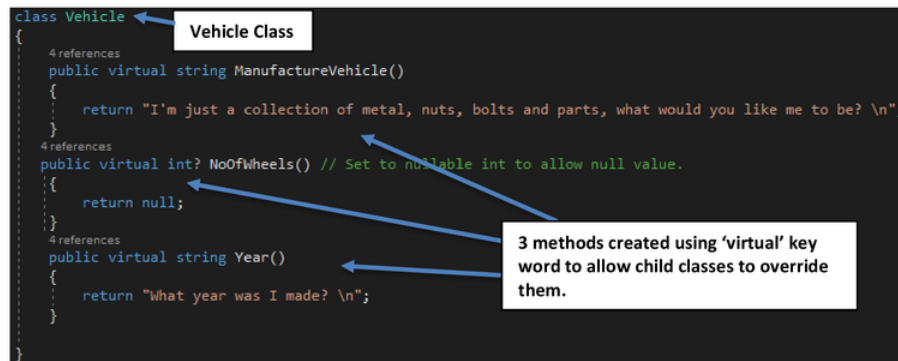


When a 'Car' object is created, it is possible to see that the 'Car' class has inherited the methods from the base 'Vehicle' class.

Polymorphism is many things in many forms, the history coming from ancient Greece where the word 'poly' means 'many' and 'Morpheus' was the Greek God of Dreams, with abilities to appear in dreams in any form. His name originating from 'morphe', the Greek word for 'form' [7], [8].

It is understood that Polymorphism allows a programmer to let different objects respond to the same method in different ways [9], it allows a derived classes version of a method to override a super classes method when called in the main program, and, therefore allowing a common interface for multiple objects whilst retaining different implementations for different objects, it allows derived classes to distinguish themselves from each other [10], [6]. Polymorphism allows programmers to implement code that can perform actions that were not conceived at the time of writing [9]. It allows different types of data to be handled by a single interface [11].

A parent 'Vehicle' class is created, containing three methods implemented using the 'virtual' keyword, each one returning string and int values. In the demonstration it is possible to see that the child classes can override these methods by using polymorphism.



```
class Vehicle
{
    4 references
    public virtual string ManufactureVehicle()
    {
        return "I'm just a collection of metal, nuts, bolts and parts, what would you like me to be? \n";
    }
    4 references
    public virtual int? NoOfWheels() // Set to nullable int to allow null value.
    {
        return null;
    }
    4 references
    public virtual string Year()
    {
        return "What year was I made? \n";
    }
}
```

Vehicle Class

3 methods created using 'virtual' key word to allow child classes to override them.

When the object is created in the main program and run it is possible to see the returned values printed to the console.

```
Vehicle vehicle = new Vehicle();
Console.WriteLine("-- MANUFACTURED VEHICLE METHOD --");
Console.WriteLine(vehicle.ManufactureVehicle());
Console.WriteLine("-- YEAR METHOD --");
Console.WriteLine(vehicle.Year());
Console.WriteLine("-- NO OF WHEELS METHOD --");
Console.WriteLine(vehicle.NoOfWheels());
```

It can be noted that there is no value for 'year' as this method has been initialised using an 'nullable int' [12].

```
-- MANUFACTURED VEHICLE METHOD --
I'm just a collection of metal, nuts, bolts and parts, what would you like me to be?
-- YEAR METHOD --
What year was I made?
-- NO OF WHEELS METHOD --
```

In the derived classes, the same methods have been created, this time using the 'override' keyword to allow them to override the virtual methods in the base class. Each one containing different return values representing attributes of each vehicle.

```
class Ship : Vehicle
{
    // Attributes
    // Constructors
    // Properties
    // Methods
    4 references
    public override string ManufactureVehicle()
    {
        return "SHIP!";
    }
    4 references
    public override int? NoOfWheels()
    {
        return 0;
    }
    4 references
    public override string Year()
    {
        return "1991";
    }
}
```

3 methods created using 'override' key word to allow them to override base class methods.

```

class Car : Vehicle
{
    // Attributes
    // Constructors
    // Properties
    // Methods
    4 references
    public override string ManufactureVehicle()
    {
        return "CAR!";
    }

    4 references
    public override int? NoOfWheels()
    {
        return 4;
    }

    4 references
    public override string Year()
    {
        return "2018";
    }
}

```

```

class Plane : Vehicle
{
    // Attributes
    // Constructors
    // Properties
    // Methods
    4 references
    public override string ManufactureVehicle()
    {
        return "PLANE!";
    }

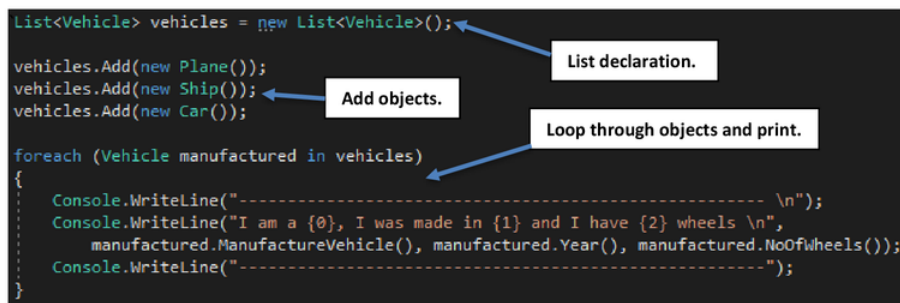
    4 references
    public override int? NoOfWheels()
    {
        return 18;
    }

    4 references
    public override string Year()
    {
        return "2005";
    }
}

```

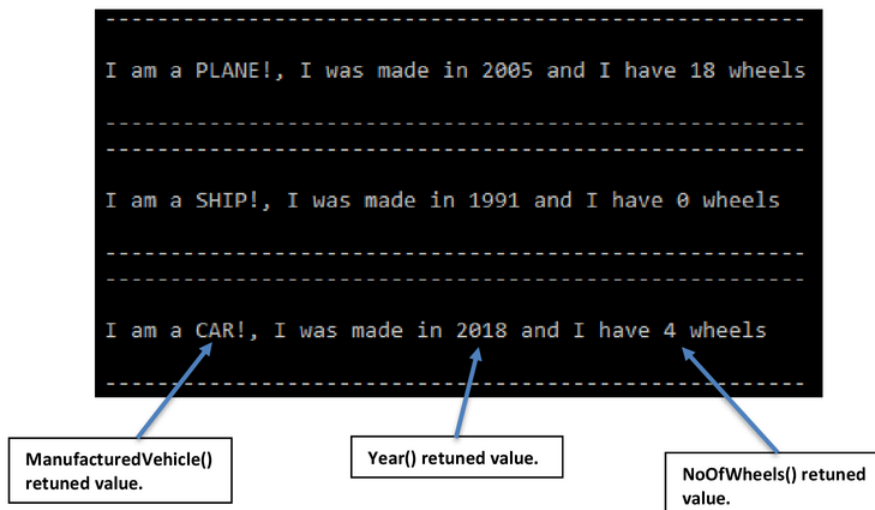
Objects can then be created for each class. For the example below, a <list> is used, with built in 'Add' method to add objects to the list of objects. A 'foreach' loop is then used to loop through each object in the list and print out the returned values.

```
List<Vehicle> vehicles = new List<Vehicle>();  
vehicles.Add(new Plane());  
vehicles.Add(new Ship());  
vehicles.Add(new Car());  
  
foreach (Vehicle manufactured in vehicles)  
{  
    Console.WriteLine("----- \n");  
    Console.WriteLine("I am a {0}, I was made in {1} and I have {2} wheels \n",  
        manufactured.ManufactureVehicle(), manufactured.Year(), manufactured.NoOfWheels());  
    Console.WriteLine("-----");  
}
```



The results show the return values in the methods in each derived class have overridden the values in the base class.

```
-----  
I am a PLANE!, I was made in 2005 and I have 18 wheels  
-----  
-----  
I am a SHIP!, I was made in 1991 and I have 0 wheels  
-----  
-----  
I am a CAR!, I was made in 2018 and I have 4 wheels  
-----
```



Polymorphism is important because object or vehicles (in shown example) come in many different shapes and forms, as demonstrated in the example above it is possible to invoke the ManufactureVehicle() method to create a vehicle irrespective of their vehicle specifics, that is to say that polymorphism allows programmers to create new types of objects without having to summon and new functionality within the new objects class. This in turn allows extendibility to existing programs; without polymorphism programs would require new methods for each new object class which was added to the program. It allows for simplified code as the client code can be unaware of what type of object it is referencing provided the class implements the methods [13], [14].

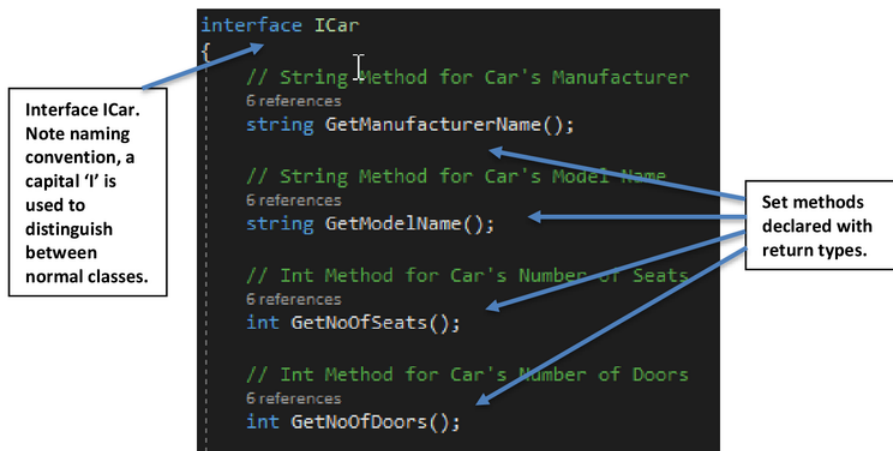


2.3 Interfaces versus Classes

In the C# programming language, inheritance from more than one class is prohibited, however, implementing multiple interfaces is permitted, this allows common functionality of interfaces across multiple classes [14]. It is understood that interfaces allow programmers to define the behaviour of an object without deciding how the object is implemented. When a class implements an interface that class must contain methods/properties which the interface needs but the class can implement these methods in different ways, it defines the capabilities of a class [1].

Interfaces are used when functionality of a class is needed for other components of a program; by using an interface instead of a class it is possible to include functionality that other classes need [15].

To demonstrate interfaces, a 'Car' interface is created containing a set of methods which will be inherited and used in three different classes; 'Sports Car', 'City Car' and 'Family Car'.



The image shows a code editor with the following C# code for an interface:

```
interface ICar
{
    // String Method for Car's Manufacturer
    6 references
    string GetManufacturerName();

    // String Method for Car's Model Name
    6 references
    string GetModelName();

    // Int Method for Car's Number of Seats
    6 references
    int GetNoOfSeats();

    // Int Method for Car's Number of Doors
    6 references
    int GetNoOfDoors();
}
```

Annotations with arrows point to specific parts of the code:

- A box on the left points to the `interface ICar` declaration: "Interface ICar. Note naming convention, a capital 'I' is used to distinguish between normal classes."
- A box on the right points to the method signatures: "Set methods declared with return types."

The methods are then called in the different car classes and return values specified. To avoid repetition of screenshots, the example class shown in the report will be the 'Sports Car' class.

```

class SportsCar : ICar
{
    // Attributes
    // Constructors
    // Properties
    // Methods

    // String Method for Car's Manufacturer
    6 references
    public string GetManufacturerName()
    {
        return "Porsche";
    }

    // String Method for Car's Model Name
    6 references
    public string GetModelName()
    {
        return "911";
    }

    // Int Method for Car's Number of Seats
    6 references
    public int GetNoOfSeats()
    {
        return 2;
    }

    // Int Method for Car's Number of Doors
    6 references
    public int GetNoOfDoors()
    {
        return 2;
    }

    // String Method for Car's Engine // From IEngine
    0 references
    public string GetEngineSize()
    {
        return "2998cc";
    }

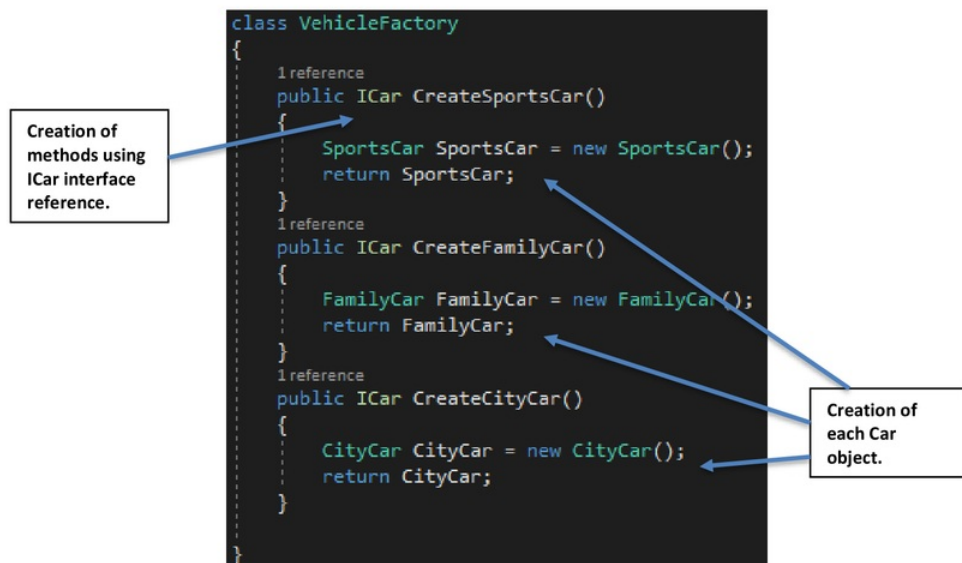
    // String Method for Engine Type // From IEngine
}

```

Indicating the SportsCar class is to inherit the ICar interface is the same syntax used when inheriting classes.

Methods declared in the interface are called in each class with different values returned.

New objects representing each of the different cars are then created in the 'Vehicle Factory' class. To create each object a method is created using the 'ICar' interface.

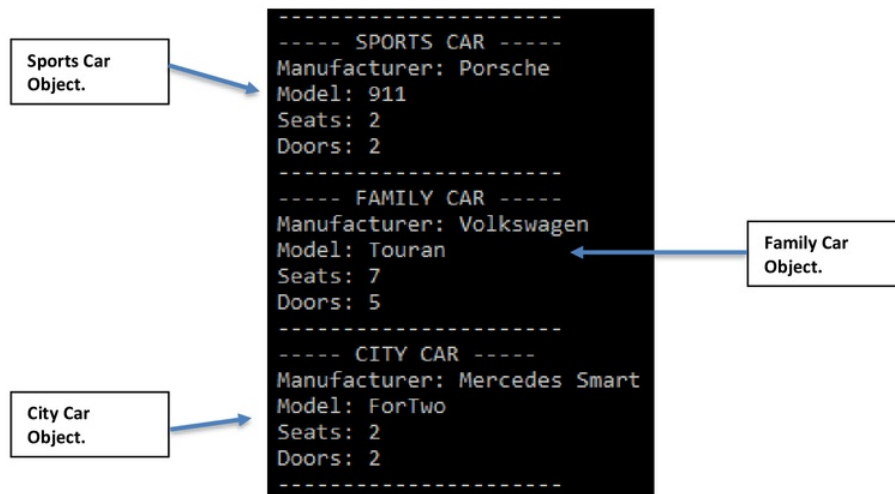


A new vehicle factory object is then created, and each car object is called. It is then possible to print out the results to show that for each car created, a different set of values have been returned for each of the methods provided by the interface. Thus making three cars which work in the same way but leaving it to different designers and manufacturers to create each car in a different ways [13].

```
static void Main(string[] args)
{
    VehicleFactory factory = new VehicleFactory();
    ICar SportsCar = factory.CreateSportsCar();
    ICar FamilyCar = factory.CreateFamilyCar();
    ICar CityCar = factory.CreateCityCar();

    Console.WriteLine(" -----");
    Console.WriteLine(" ---- SPORTS CAR ----");
    Console.WriteLine(" Manufacturer: {0}", SportsCar.GetManufacturerName());
    Console.WriteLine(" Model: {0}", SportsCar.GetModelName());
    Console.WriteLine(" Seats: {0}", SportsCar.GetNoOfSeats());
    Console.WriteLine(" Doors: {0}", SportsCar.GetNoOfDoors());
    Console.WriteLine(" -----");
    Console.WriteLine(" ---- FAMILY CAR ----");
    Console.WriteLine(" Manufacturer: {0}", FamilyCar.GetManufacturerName());
    Console.WriteLine(" Model: {0}", FamilyCar.GetModelName());
    Console.WriteLine(" Seats: {0}", FamilyCar.GetNoOfSeats());
    Console.WriteLine(" Doors: {0}", FamilyCar.GetNoOfDoors());
    Console.WriteLine(" -----");
    Console.WriteLine(" ---- CITY CAR ----");
    Console.WriteLine(" Manufacturer: {0}", CityCar.GetManufacturerName());
    Console.WriteLine(" Model: {0}", CityCar.GetModelName());
    Console.WriteLine(" Seats: {0}", CityCar.GetNoOfSeats());
    Console.WriteLine(" Doors: {0}", CityCar.GetNoOfDoors());
    Console.WriteLine(" -----");
}
```

Each method is called whilst specifying the different object, printing out the different values.



As previously mentioned, C# does not support inheriting multiple classes into one class, but interfaces can be inherited. To demonstrate this, a separate IEngine interface is created to contain more specific methods relating to the car's engine.

```
interface IEngine
{
    // String Method for Engine Size
    // References
    string GetEngineSize();

    // String Method for Engine Type
    // References
    string GetEngineType();

    // String Method for Engine Fuel Type
    // References
    string GetFuelType();
}
```

This interface is then inherited to each class. Interfaces are separated with a comma. It is now compulsory that the methods declared in IEngine are used in each class.

```
class SportsCar : ICar, IEngine
{
```

```

// String Method for Car's Engine // From IEngine
1reference
public string GetEngineSize()
{
    return "2998cc";
}
// String Method for Engine Type // From IEngine
1reference
public string GetEngineType()
{
    return "Boxer Engine";
}
// String Method for Engine Fuel Type
1reference
public string GetFuelType()
{
    return "Petrol";
}

```

The newly inherited interface classes created and assigned return values. New sports car engine objects are then created in the vehicle factory class and then called in the main program.

```

public IEngine CreateSportsCarEngine()
{
    SportsCar SportsCarEngine = new SportsCar();
    return SportsCarEngine;
}
1reference
public IEngine CreateFamilyCarEngine()
{
    FamilyCar FamilyCarEngine = new FamilyCar();
    return FamilyCarEngine;
}
1reference
public IEngine CreateCityCarEngine()
{
    CityCar CityCarEngine = new CityCar();
    return CityCarEngine;
}

```

```

IEngine SportsCarEngine = factory.CreateSportsCarEngine();
IEngine FamilyCarEngine = factory.CreateFamilyCarEngine();
IEngine CityCarEngine = factory.CreateCityCarEngine();

```

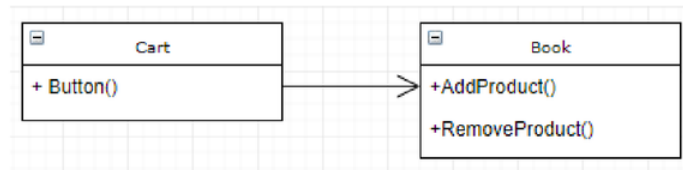
These values can then be printed out with the existing results to show that each class is inheriting from the different interfaces.

```
-----  
----- SPORTS CAR -----  
Manufacturer: Porsche  
Model: 911  
Engine Size: 2998cc  
Engine Size: Boxer Engine  
Engine Size: Petrol  
Seats: 2  
Doors: 2  
-----  
----- FAMILY CAR -----  
Manufacturer: Volkswagen  
Model: Touran  
Engine Size: 1998cc  
Engine Size: Inline four Cylinder  
Engine Size: Diesel  
Seats: 7  
Doors: 5  
-----  
----- CITY CAR -----  
Manufacturer: Mercedes Smart  
Model: ForTwo  
Engine Size: 800cc  
Engine Size: Inline Three Cylinder  
Engine Size: Petrol  
Seats: 2  
Doors: 2  
-----
```

2.4 Benefits & Drawbacks of Dependency Inversion Principle

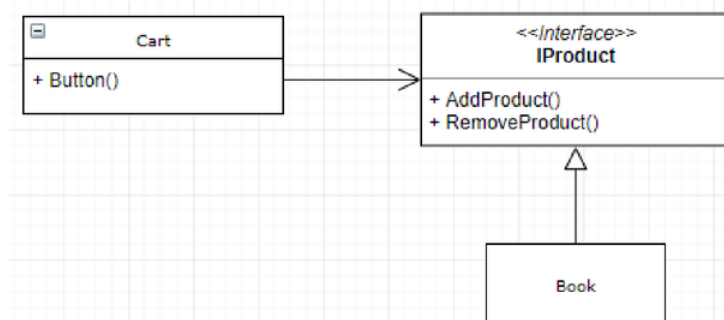
Dependency inversion principle, first introduced conceptually by Robert C. Martin in 1996 [16] states that *"high level modules (classes) should not depend upon low level modules, both should depend on abstractions and that abstractions should not depend upon details, details should depend upon abstractions"*[16]–[19]. That is to say that base classes should not be depended on derived classes, nor should derived classes depend on base classes, both should depend on a common abstraction.

An example of where dependency is inverted can be seen later in this assignment in Exercise 3 where the program contains a 'Cart' class, three product classes and a product interface. For this example, the 'Book' class will be used to demonstrate the DIP. The following example is modified from an example shown by R. Martin [19].



The 'Cart' object receives messages from a button and sends requests to either add or remove a book to or from the cart. This implies that the higher-level module 'Cart' depends on the lower level module 'Book'. This means that 'Cart' will be affected by changes to the 'Book' class.

By introducing an abstraction into the program, the Cart's dependency on 'Book' is removed. And in turn, 'Book' is no longer dependant on cart. Both are now depend on the IProduct abstraction.



An interface is added to break up the direct relationship and direct dependency of 'Cart' on the 'Book'. It is now irrelevant what product can be added to the cart as many more objects can be created to implement the IProduct interface.

The benefits of dependency inversion are clear from the example, and backed up by Hunt & Thomas in their book 'The Pragmatic Programmer' where they argue that using the DIP pattern a more robust and flexible program is created, which allows programmers to reuse certain areas of the program with different data and allows customisation of code without rebuilding [20]. These benefits are also mentioned in other researched readings [21]–[23].

The DIP, however beneficial does come with its own specific drawbacks, most notably, that the testing of the program and its modules becomes more difficult [24]. It is also considered that code is more difficult to read and to follow due to all the modules being split up by different abstractions.

The DIP can be known as the Abstract Factory Pattern, as stated by Freeman et al in their 2009 publication "Head First Design Patterns" [25].

Software Testing

Software testing is an integral part of any software project. According to the International Software Testing Qualifications Board (ISTQB) it is the process of running a program with the purpose of finding bugs, also referred to as validation and verifying of a program to ensure that it meets certain criteria and works as expected [26].

Given how important software testing is to the industry; 50% of time and 50% of budget are expended in testing [27], it could be perceived that software testing is an exact science, however, Myers states in his 2012 book 'The Art of Software Testing' that *"less seems to be known about software testing than any other aspect of software development"* he also describes it as an *"dark art"* [27]. This is evident in recent projects with so many software companies falling foul of poor testing, most notable examples being the NHS booking system, HMRC's faulty check payment system and Amazon's glitch allowing customers to purchase items for 1 pence [28]–[31].

There are two main types of software testing, Black Box Testing and White Box testing. These types of testing are also known as 'Functional Testing' or 'Input/output Testing' and 'Structural Testing' or 'Glass Box Testing' respectively [26][27]. However, there are also many other types of software testing including; Unit Testing, Integration Testing, Functional Testing, System Testing, Stress Testing, Performance Testing, Usability Testing, Acceptance Testing, Regression Testing and Beta Testing [26], [32], [33], to name a few. This assignment will discuss Black Box, White Box and Unit Testing (form of white box testing).

3.1 Black Box Testing

Black Box testing is a method of software testing that overlooks the internal mechanism or structure of the software, with focus on testing with many different inputs and confirming results against anticipated output [26], [27], [34], hence the name 'Input/Output Testing'. The definition by the ISTQB; *"Testing, either functional or non-functional, without reference to the internal structure or system"*[35], black box testing is therefore more end-user experience focused. Later in this assignment, boundary testing, a form of Black Box testing [36], will be used to test the outputs of exercise 2 & 3.

3.2 White Box Testing

The technique of White Box testing tests beyond the user interface [37], it uses knowledge of the internal workings of a program in order to select different test data. This method uses detailed knowledge of the software code to scrutinise the outputs from the program, which renders the test useless unless the tester is aware of what the program has been designed to do [27], [33], [38]. In order to complete White Box tests, programming knowledge is crucial, however, it is understood from lectures that there is a danger tests become biased due to a programmer adapting tests to suit each program. The ISTQB define White Box testing as *"Testing based on an analysis of the internal structure of the component or system"* [39].

Black Box testing is frequently used for validation whereas White Box testing is frequently used for verification. It is said that both tests must be performed on software for a complete examination [33], [38].

3.3 Unit Testing

Unit or module testing is the when individual subprograms/units/modules/classes of a program are tested individually, it tests the smaller pieces of the software over the software as a whole, it generally follows the White Box testing methodology as it is a form of White Box testing. It allows the tester to determine an output methods and procedures individually, in turn allowing better management of bug finding, furthermore it allows a catalogue of tests to be compiled which can be run as the software develops to continually check program quality [27], [40], [41].

Object Orientated Programming

4.1 Exercise One

- Build a program that uses a class to count the number of spoons that have been manufactured by a machine. Each spoon has a manufacture cost of 0.20p

Version One

To implement the program asked, firstly a class 'Counter' was created to contain the attributes, constructors, properties and methods required for creating a counting program.

// Attributes

```
private int _totalSpoons;  
private double _spoonCost;
```

Attributes declared to allow user to get and set values.

Constructor method is then created to initialise the data members of the object 'Counter' when it is created later in the program.

// Constructors

```
1 reference  
public Counter(int totalSpoons, double spoonCost)  
{  
    _totalSpoons = totalSpoons;  
    _spoonCost = spoonCost;  
}
```

// Properties

```
1 reference  
public void SetSpoons(int spoonTotal)  
{  
    _totalSpoons = spoonTotal;  
}  
2 references  
public int GetSpoonTotal()  
{  
    return _totalSpoons;  
}
```

Property methods (accessors) are then created to allow user to 'get' and 'set' values when creating the counter object.

The method 'TotalSpoonCost()' is created to return the total number of spoons manufactured multiplied by the spoon cost.

```
public double TotalSpoonCost()  
{  
    return (_totalSpoons * _spoonCost);  
}
```

In the main program, a new object is then created called 'CountSpoons' the totalSpoon value set to 0 and spoonCost set to 0.20p.

```
// Set value of spoonTotal to 0 and set value of spoonCost to 0.20p.
Counter CountSpoons = new Counter(0, 0.20);
```

The total number of spoons is then set. The total number of spoons is then printed out in the console along with the total cost.

```
CountSpoons.SetSpoons(1);
Console.WriteLine(CountSpoons.GetSpoonTotal());
Console.WriteLine(CountSpoons.TotalSpoonCost());
```

Results from the console shown below.

```
1
0.2
_
```

4.1.1 Exercise One Testing

Black Box Test on Version 1

Test	Steps	Expected Result	Actual Result
Program prompts user to enter number of spoons manufactured.	1 – Open program	Program displays message in console asking user to enter number.	Program displays total spoons declared and total price of spoons.
Program displays number of spoons entered and total cost of spoons in £'s & pence.	1 – Enter amount of spoons produced into program numerically. 2 – Press return.	Program displays number of spoons entered and total cost of spoons in £'s & pence.	N/A
Close Program	1 – Hit return to close program.	Program Closes	Program Closes.

Above test shows that although the program is working, the program should be able to allow user to enter the number of spoons to be calculated, rather than being set by the programmer. To allow this, the following code has been added and the previously set Values for 'CountSpoon' object are removed.

The user prompted to enter the number of spoons manufactured, this is then read by the program and assigned to variable 'machineTotal', this is then added to the total of spoons already assigned to the 'CountSpoons' object, (0). The total entered is then printed out to the console.

```
// Prompt user to enter spoons manufactured
Console.Write("Enter Total Spoons Manufactured by Machine: ");

// Read amount of spoons entered and add to count of spoons set to 0 upon creation of spoon counter.
int machineTotal = int.Parse(Console.ReadLine());
CountSpoons.SetSpoons(CountSpoons.GetSpoonTotal() + machineTotal);

Console.WriteLine("");
// Display total amount of spoons entered.
Console.WriteLine("Total Spoons Manufactured: {0} \n", CountSpoons.GetSpoonTotal());
```

The TotalSpoonCost() method is then called from the 'CountSpoons' object and the total is displayed.

```
// Display Total Cost of Spoons
Console.WriteLine(" Total Cost of Spoons: £{0}", CountSpoons.TotalSpoonCost());
```

Results from the console.

```
-----
Enter Total Spoons Manufactured by Machine: 9

Total Spoons Manufactured: 9

Total Cost of Spoons: £1.8
-----
```

Black Box Version 2 Test

Test	Steps	Expected Result	Actual Result
Program prompts user to enter number of spoons manufactured.	1 – Open program	Program displays message in console asking user to enter number.	Program displays message in console asking user to enter number.
Program displays number of spoons entered and total cost of spoons in £'s & pence.	1 – Enter amount of spoons produced into program numerically. 2 – Press return.	Program displays total number of spoons entered and total cost calculation in format of £'s & pence.	Program displays total number of spoons entered and total cost calculation but in wrong format.
Close Program	1 – Hit return to close program.	Program Closes	Program Closes.

Version 2 test show that the program now fulfils it desired purpose, however, the total cost of spoons is displayed in the wrong format.

To return the value of 'TotalSpoonCost' in the console, in the correct format, the method 'CostFormatter' is created to format the double value into a string.

```
public string CostFormatter(double formattedTotal)
{
    string formattedTotalCost = String.Format("{0:0.00}", formattedTotal);
    return formattedTotalCost;
}
```

A variable is then created to store the returned value from method 'TotalSpoonCost', this is then passed into the 'CostFormatter()' method when called on the CountSpoons object to convert the double value into a string to display the total cost in the correct price format.

```
// Assign TotalSpoonCost method's returned value into variable 'totalSpoonCost' to pass to string formatter method CostFormatter.  
double totalSpoonCost = CountSpoons.TotalSpoonCost();  
Console.WriteLine("Total Manufactured Cost of Spoons: £(0)p", CountSpoons.CostFormatter(totalSpoonCost));  
Console.WriteLine("-----");  
Console.ReadKey();
```

Testing of the program with 1 spoon yields the following results.

```
-----  
Enter Total Spoons Manufactured by Machine: 1  
Total Spoons Manufactured: 1  
Total Manufactured Cost of Spoons: £0.20p  
-----
```

Testing with 10 spoons.

```
-----  
Enter Total Spoons Manufactured by Machine: 10  
Total Spoons Manufactured: 10  
Total Manufactured Cost of Spoons: £2.00p  
-----
```

Black Box Version 3 Test

Test	Steps	Expected Result	Actual Result
Program prompts user to enter number of spoons manufactured.	1 – Open program	Program displays message in console asking user to enter number.	Program displays message in console asking user to enter number.
Program displays number of spoons entered and total cost of spoons in £'s & pence.	1 – Enter amount of spoons produced into program numerically. 2 – Press return.	Program displays total number of spoons entered and total cost calculation in format of £'s & pence.	Program displays total number of spoons entered and total cost calculation in format of £'s & pence.
Close Program	1 – Hit return to close program.	Program Closes	Program Closes.

The program now fulfils all expected criteria.

4.2 Exercise Two

- Develop a class that represents a circle that exists in 2D space at a given set of coordinates (x, y) with a radius R. Add a method to the Circle class that tests whether another circle overlaps with it.

A circle class is developed to allow a circle object to be created. Firstly, attributes for the X and Y coordinates are created along with R for the radius.

```
// Attributes
private int _x;
private int _y;
private double _r;
```

A constructor is then created for the Circle object passing in the integer values for x & y, and double value for radius.

```
// Constructors
2 references
public Circle(int x, int y, double r)
{
    _x = x;
    _y = y;
    _r = r;
}
```

Properties are declared to get the values.

```
// Properties
2 references
public int GetPointX()
{
    return _x;
}
2 references
public int GetPointY()
{
    return _y;
}
2 references
public double GetRadius()
{
    return _r;
}
```

A Boolean method is then created to calculate the distance between the two points (centre of each circle) on their x & y axis. The method also contains a calculation to sum the two radii values. If the distance between the two centres of the circles is known, and the total length of the radii summed is known then it is possible to conclude that if the sum of the radii is greater than the total length between the two centres of the circles then the two circles must be overlapping. Consequently, if the sum of the radii is equal to that of the distance between x & y then the circles are touching, if the distance between x & y is greater than the sum of the radii, the circles are not touching.

To work out the distance between the centres of the two circles, it is best to think of the points at the centre of each circle making up the points of a triangle. This can be seen in the fig 1 below.

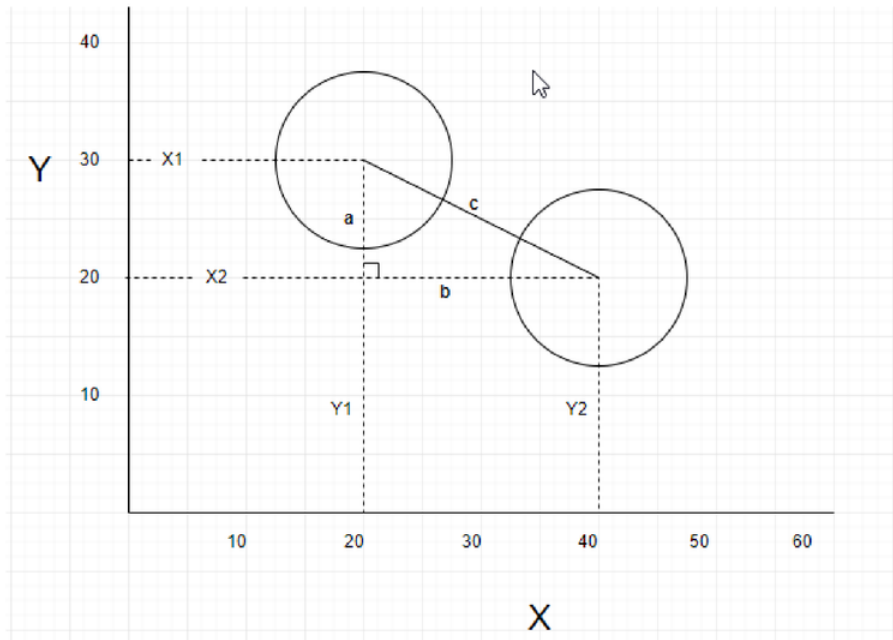


Fig 1. Demonstration of hypothetical right handed triangle.

It is then possible to work out the length values for **a** & **b** using the following sums.

$$\begin{aligned}x1 - x2 &= a \\y1 - y2 &= b\end{aligned}$$

Once values for **a** & **b** are known, it is then possible to use Pythagoras' theorem to calculate the hypotenuse (distance between the centres of the circles) of the triangle.

$$c = \sqrt{a^2 + b^2}$$

Now that the distance between the centre circles & the summed radii is known an 'if-statement' is used to compare the values and true or false values are returned.

```
// Methods
1 reference
public bool CheckOverlap(Circle circle) // Pass in existing Circle object.
{
    int a = _x - circle.GetPointX(); // Get in properties is acting as second circle.
    int b = _y - circle.GetPointY();
    double sumRadii = _r + circle.GetRadius();
    double hypotenuse = Math.Sqrt(a * a + b * b);

    if (sumRadii > hypotenuse)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

The method above takes the existing values from the circle object C1 (created in the main program) and then by passing in another circle object (C2), it can perform the equation using the different circles coordinates. Where C1 coordinates are `_x` & `_y`, and C2 coordinates are `circle.GetPointX()` & `circle.GetPointY()` respectively.

In the main program the user is prompted to enter values for the x, y and r values. These are stored in variables `pointX`, `pointY` and `radius` and passed into the circle objects C1 & C2 upon their creation.

```
// Prompt user to enter Circle One's Values
Console.WriteLine(" -----");
Console.Write(" Enter Circle One X Value: ");
// Read Value X and store in variable pointX.
int pointX1 = int.Parse(Console.ReadLine());
// Enter and Read Value Y and store in variable pointY
Console.Write(" Enter Circle One Y Value: ");
int pointY1 = int.Parse(Console.ReadLine());
// Enter and Read Radius Value
Console.Write(" Enter Circle One Radius: ");
double radius1 = double.Parse(Console.ReadLine());
Console.WriteLine(" -----");

// Prompt user to enter Circle Two's values
Console.Write(" Enter Circle Two X Value: ");
// Read Value X and store in variable pointX.
int pointX2 = int.Parse(Console.ReadLine());
// Enter and Read Value Y and store in variable pointY
Console.Write(" Enter Circle Two Y Value: ");
int pointY2 = int.Parse(Console.ReadLine());
//Enter and Read Radius Value
Console.Write(" Enter Circle Two Radius: ");
double radius2 = double.Parse(Console.ReadLine());
Console.WriteLine(" ----- \n");

// Set value of Circle1 x, y, r passing in values user has entered
Circle C1 = new Circle(pointX1, pointY1, radius1);
// Set value of Circle2 x, y, r
Circle C2 = new Circle(pointX2, pointY2, radius2);
```

An 'if-statement' is then used to call the 'CheckOverlap()' method with circles C1 & C2 passed in whilst comparing the return value to true. If the value returned is true a message is displayed to the user to say the circles are overlapping. If the returned value from the 'CheckOverlap()' isn't true, a message is displayed conveying the circles do not overlap.

```
if (C1.CheckOverlap(C2) == true)
{
    Console.WriteLine(" CIRCLES OVERLAPPING!");
}
else
{
    Console.WriteLine(" CIRCLES NOT OVERLAPPING!");
}
```

4.2.1 Exercise Two Testing

Black Box Boundary Test Coordinate Set 1

Test	Steps	Expected Result	Actual Result
Program prompts user to enter X^1 , Y^1 , R^1 , X^2 , Y^2 & R^2 coordinates.	1 – Open program. 2 – Enter $X^1 = 20$, $Y^1 = 30$, $R^1 = 10.5$, $X^2 = 40$, $Y^2 = 20$ & $R^2 = 10.5$ coordinates.	Program stores values and calculates if circles are overlapping. Circles should not overlap.	Success - Circles do not overlap.
Close Program	1 – Hit return to close program.	Program Closes	Program Closes.

Testing the program with above coordinates shows that program is working correctly. This can be visualised by comparing with fig 2 below. Results of the console are also shown below.

```
-----
Enter Circle One X Value: 20
Enter Circle One Y Value: 30
Enter Circle One Radius: 10.5
-----
Enter Circle Two X Value: 40
Enter Circle Two Y Value: 20
Enter Circle Two Radius: 10.5
-----
CIRCLES NOT OVERLAPPING!
```

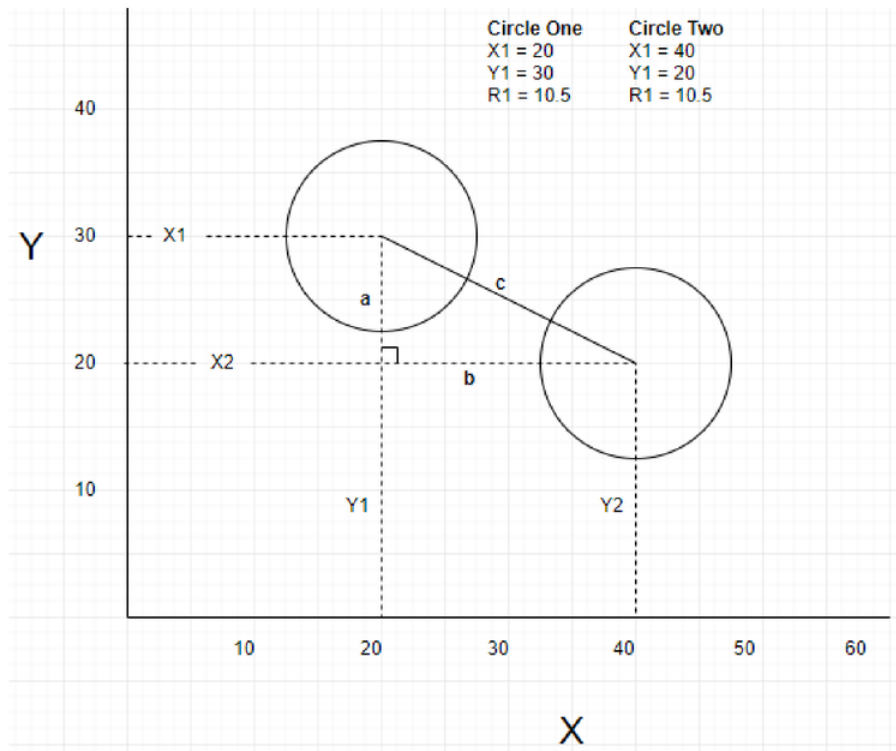


Fig 2. Exact coordinates used in testing set 1. Circles not overlapping.

Black Box Boundary Test Coordinate Set 2

Test	Steps	Expected Result	Actual Result
Program prompts user to enter X^1 , Y^1 , R^1 , X^2 , Y^2 & R^2 coordinates.	1 – Open program. 2 – Enter $X^1 = 20$, $Y^1 = 30$, $R^1 = 20.2$, $X^2 = 40$, $Y^2 = 20$ & $R^2 = 20.4$ coordinates.	Program stores values and calculates if circles are overlapping. Circles should not overlap.	Success - Circles overlap.
Close Program	1 – Hit return to close program.	Program Closes	Program Closes.

Increasing the radius size of the two circles to test the overlapping condition. Fig 3 demonstrates on a graph.

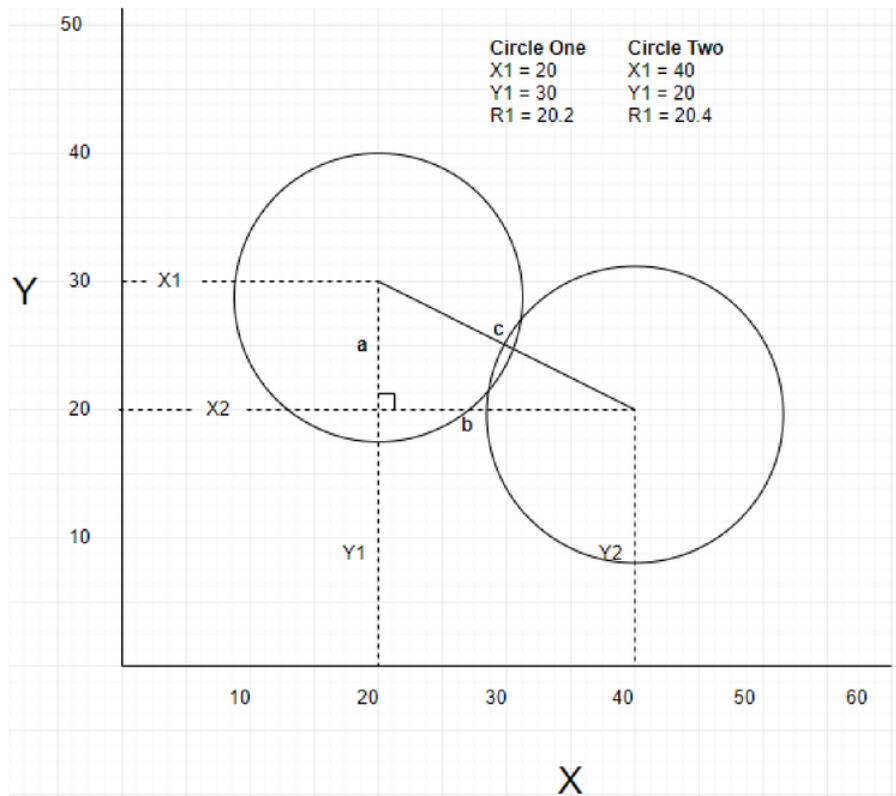


Fig 3. Increased radii size, visualising circles overlapping.

Screenshot below shows results of boundary test two.

```

-----
Enter Circle One X Value: 20
Enter Circle One Y Value: 30
Enter Circle One Radius: 20.2
-----
Enter Circle Two X Value: 40
Enter Circle Two Y Value: 20
Enter Circle Two Radius: 20.4
-----
CIRCLES OVERLAPPING!

```

Sketches and workings are provided in the appendix of this report to show how calculations and logic were implemented.

4.3 Exercise Three

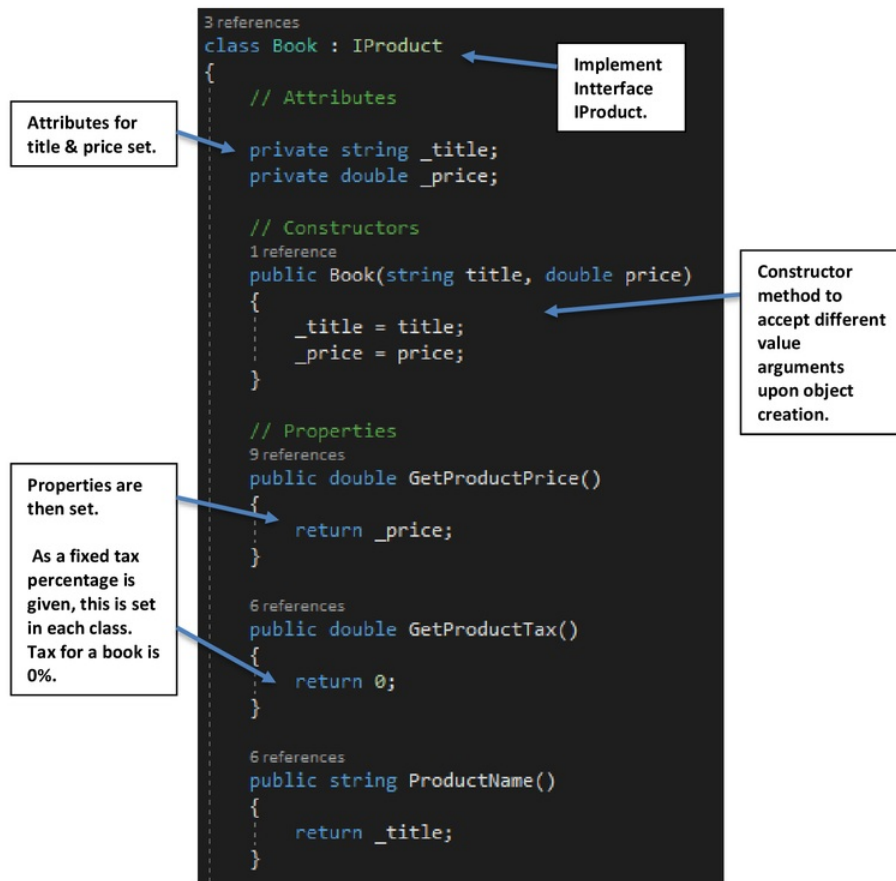
- Create an OOP program, with a shopping cart class, that can contain a number of different products being purchased. The price of each product varies but the tax due depends on the product type.

From following examples shown in lectures, it was recognised that this exercise would be solved using a common interface to implement the same methods across each product class. The three common pieces of information required are, product price, product name & product tax. Three methods were created in the interface 'IProduct' to represent these values with correct return types.

```
10 references
interface IProduct
{
    // Product Name Method
    6 references
    string ProductName();
    // Product Price Method
    9 references
    double GetProductPrice();
    // Product Tax Method
    6 references
    double GetProductTax();
}
```

Four classes are then created, one to represent the shopping cart and three to represent products; CD, Book & DVD.

In each product class contains attributes to create the product title & price. For simplicity in the example, each different product class contains the same attributes. The 'Book' class is shown and annotated below.



The shopping cart class contains one attribute, this time, a <List> is used named ‘_cartContents’. A list allows objects to be added to it, which can be accessed by index [42]. It is thought that a list is like an array for objects. The IProduct class is implemented into the list to allow each method and value to be added with each object.

```

// Attributes
// Create list to add products to in the shopping cart.
private List<IProduct> _cartContents = new List<IProduct>();

```

Five methods are created in the cart class. The first is to add a product to the cart list.

```

// Methods
// Add products to list of products in the shopping cart.
3 references
public void AddProduct(IProduct item)
{
    _cartContents.Add(item);
}

```

A method to calculate the subtotal is then implemented, simply adding up the cost of each product before tax. A 'foreach' loop is used to iterate through each product in the list.

```
1 reference
public double SubTotalProductCost()
{
    // Set total of cart to 0. // Outside foreach to avoid setting back to 0 everytime.
    double subTotal = 0;
    // Loop through each product in cart and add cost of items before tax.
    foreach (IProduct product in _cartContents)
    {
        subTotal = subTotal + product.GetProductPrice();
    }
    //return grandTotal;
    return subTotal;
}
```

The next two methods calculate the tax. The first calculates the total tax percentage which will be added to the sub total of products.

```
1 reference
public double TotalProductTax()
{
    double tax = 0;
    // Loop through each product in cart and add total tax percentage to be added to cart cost.
    foreach (IProduct product in _cartContents)
    {
        tax = tax + product.GetProductTax();
    }
    return tax;
}
```

The second, calculates the total tax monetary value to be added to the subtotal to gain the grand total.

```
1 reference
public double TotalProductTaxCost()
{
    double subtotal = 0;
    double tax = 0;
    double totalTax = 0;
    // Loop through each product in cart and work out total cost of tax on items in cart.
    foreach (IProduct product in _cartContents)
    {
        subtotal = subtotal + product.GetProductPrice();
        tax = tax + product.GetProductTax();
        totalTax = subtotal * tax / 100;
    }
    //return grandTotal;
    return totalTax;
}
```

Finally, the total tax cost and subtotal are summed and returned.

```

public double TotalProductCost()
{
    // Set total of cart to 0. // Outside foreach to avoid setting back to 0 everytime.
    double subtotal = 0;
    double tax = 0;
    double totalTax = 0;
    double grandTotal = 0;
    // Loop through each product in cart and add cost of all products and tax and return grand total.
    foreach (IProduct product in _cartContents)
    {
        subtotal = subtotal + product.GetProductPrice();
        tax = tax + product.GetProductTax();
        totalTax = subtotal * tax / 100;
        grandTotal = subtotal + totalTax;
    }
    return grandTotal;
}

```

A Cart object is then created in the main program to allow product objects to be added to its list. Three products are also created to add to the cart, with title name and price passed in.

```

// Create cart.
Cart Cart = new Cart();
// Create Book.
Book Book = new Book("Da Vinci Code", 1.99);
// Add book to cart.
Cart.AddProduct(Book);

// Create DVD.
DVD DVD = new DVD("Kids", 8.99);
// Add DVD to cart.
Cart.AddProduct(DVD);

// Create CD/
CD CD = new CD("Dr. Dre 2001", 3.99);
// Add CD to cart
Cart.AddProduct(CD);

```

The methods ProductName() & ProductPrice() are then called and wrapped in a 'Console.WriteLine' to display each product to display each product in the cart.

```

Console.WriteLine(" ----- CART CONTENTS ----- ");
Console.WriteLine(" {0} £{1}", Book.ProductName(), Book.GetProductPrice());
Console.WriteLine(" {0} £{1}", DVD.ProductName(), DVD.GetProductPrice());
Console.WriteLine(" {0} £{1}", CD.ProductName(), CD.GetProductPrice());
Console.WriteLine(" -----");

```

A complete price breakdown is then provided by calling each corresponding method in the cart class and printing out to the console.

```

Console.WriteLine(" Sub Total: £{0}", Cart.SubTotalProductCost());
Console.WriteLine(" Total Tax: {0}%", Cart.TotalProductTax());
Console.WriteLine(" Total Tax: £" + string.Format("{0:0.00}", Cart.TotalProductTaxCost()));
Console.WriteLine(" Grand Total: £" + string.Format("{0:0.00}", Cart.TotalProductCost()));
Console.WriteLine(" -----");

```

A 'string.Format' built in method is used to convert the double value to a string and format the returned value to be rounded to 2 decimal places.

4.3.1 Black Box Boundary Testing – Test One

Test	Steps	Expected Result	Actual Result
Program Displays items in shopping cart and provides total cost breakdown in console. <u>Test One Items</u> Book £1.99 DVD £8.99 CD £3.99	1 – Open program.	Sub Total: £14.97 Tax: 22% Total Tax: £3.29 Grand Total: £18.26	Success – Correct calculations from program.
Close Program	1 – Hit return to close program.	Program Closes	Program Closes.

Results from the console are shown below.

```

----- CART CONTENTS -----
Da Vinci Code £1.99
Kids £8.99
Dr. Dre 2001 £3.99
-----
Sub Total: £14.97
Total Tax: 22%
Total Tax: £3.29
Grand Total: £18.26
-----

```

Black Box Boundary Testing – Test Two

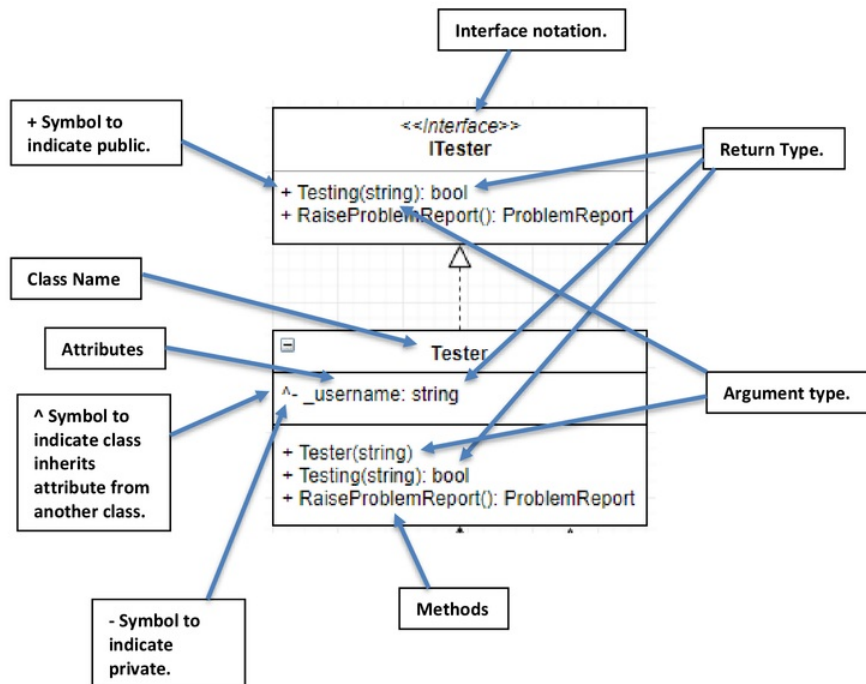
Test	Steps	Expected Result	Actual Result
Program Displays items in shopping cart and provides total cost breakdown in console. <u>Test One Items</u> Book £11.99 DVD £18.99 CD £2.99	1 – Open program.	Sub Total: £24.97 Tax: 22% Total Tax: £5.49 Grand Total: £30.46	Success – Correct calculations from program.
Close Program	1 – Hit return to close program.	Program Closes	Program Closes.

```
----- CART CONTENTS -----  
C# for Dummies £11.99  
The Deer Hunter £8.99  
Britney Spears £3.99  
-----  
Sub Total: £24.97  
Total Tax: 22%  
Total Tax: £5.49  
Grand Total: £30.46  
-----
```

UML Diagram

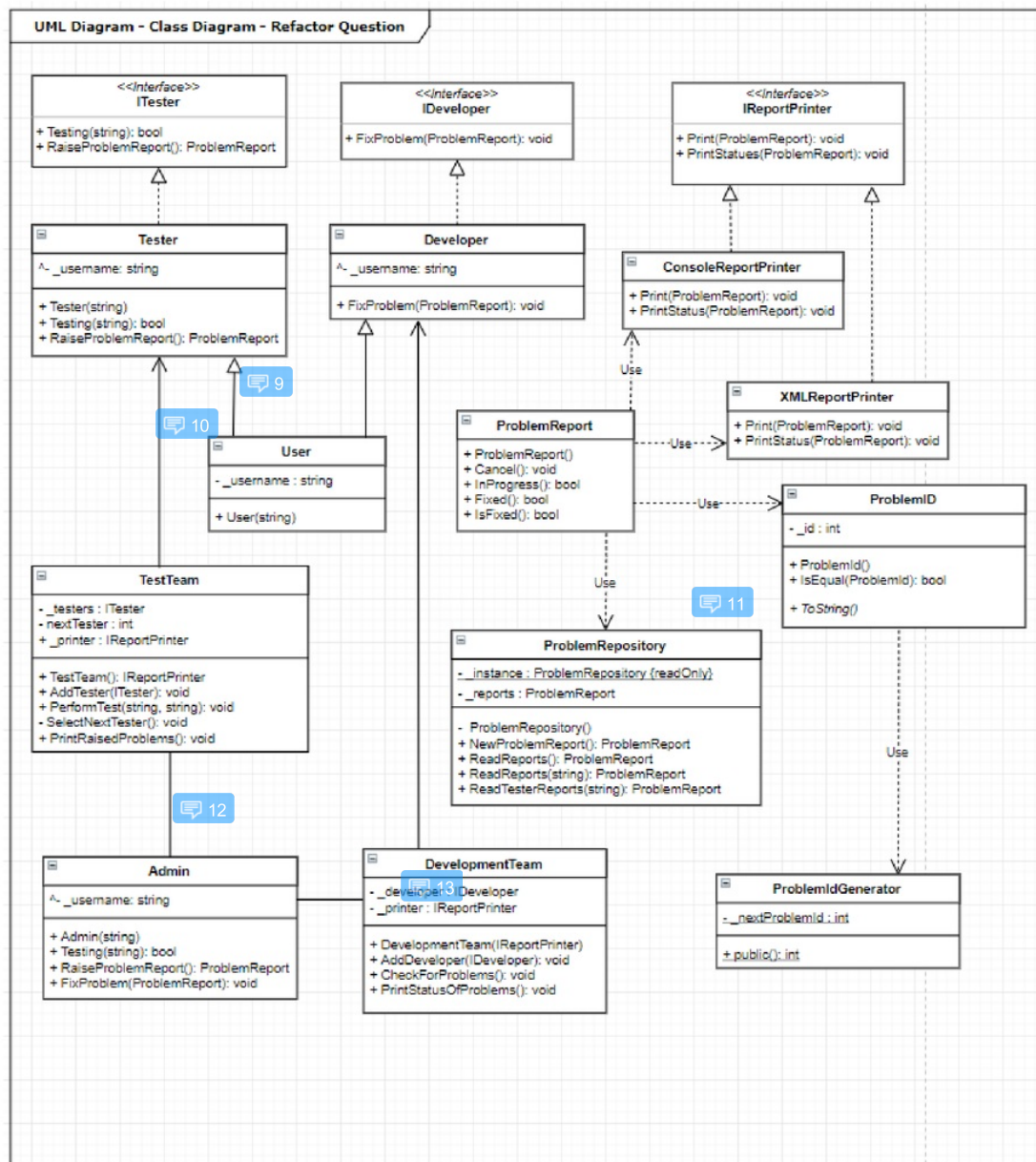
5.1 Class Diagram

Source code for an application has been provided to be translated into a UML Class diagram. Firstly a guide of class diagram objects and notations will be provided.



The complete class diagram is shown overleaf. Disclaimer; Henry Cobbold and the author of this report discussed methodologies, and researched class diagrams together to further our understanding on the diagrams, as a result, the class diagram produced may be similar in nature to that of Mr. Cobbold's. The author can confirm that although research was carried out together, the UML diagram produced was that of his own work.

8



In the appendix of this report a table is shown which was drawn up and used to aid in the design of the class diagram and its relationships.

Critical Reflection

Object orientated programming concepts have presented new challenges, most of which have been difficult to understand and implement. The research questions at the beginning of this assignment proved very helpful in aiding understand of the basic principles of OOP, however, polymorphism & the dependency inversion principle are areas which need more research.

The programming exercises also provided a new challenge, working with classes and objects in C# is a new skill which needs to be practiced. Exercise two, for example, could of be improved if extra functionality had been implemented to show whether the circles were touching as well as overlapping or not overlapping. Similarly, in exercise three, it is felt that adding products to the cart could have been implemented via the console, where users could select a product to add to the cart instead of the items being hard coded in the code.

Features such as the one desired in exercise three will require more practice and research. A simple improvement like that described for exercise two could be implemented with better time management of this assignment. Exercise four was not attempted, again, more research and practice with programming is needed.

The testing process also suffered from time constraints, it had been planned to perform more tests boundary tests on each program, and attempt white box unit tests using the built in testing feature on Visual Studio [43]. The programming exercises were also attempted, and some completed before research was done into testing, this hindered the testing as it would have been beneficial to develop the programs through testing, a process which will be implemented in the future.

The sequence diagram was also omitted from the assignment, again, due to time constraints as more research was needed into the sequence diagrams before an attempt was made. Less focus on other areas of the assignment could of allowed more research into this area.

The assignment on a whole has proved intellectually challenging and has pushed forward theoretical and practical knowledge, it also has provided useful revision for the upcoming exam.

References

- [1] R. Miles, *C# Programming Yellow Book*, no. November. 2015.
- [2] UCAR, "Understanding classes and objects." [Online]. Available: https://www.ncl.ucar.edu/Document/HLUs/User_Guide/classes/classoview.shtml. [Accessed: 12-Mar-2018].
- [3] M. O' Docherty, *Object-Oriented Analysis and Design: Understanding System Development with UML 2.0*. John Wiley & Sons, 2005.
- [4] J. L. Popyack, "Object Oriented Programming," 2015. [Online]. Available: https://www.cs.drexel.edu/~introcs/Fa15/notes/06.1_OOP/Advantages.html?CurrentSlide=3. [Accessed: 12-Mar-2018].
- [5] M. Yaiser, "Object-oriented programming concepts: Inheritance | Adobe Developer Connection," 2012. [Online]. Available: <https://www.adobe.com/devnet/actionscript/learning/oop-concepts/inheritance.html>. [Accessed: 13-Mar-2018].
- [6] Department of Computer Science and Calvin College, "Inheritance and Polymorphism."
- [7] "Poly- | Define Poly- at Dictionary.com." [Online]. Available: <http://www.dictionary.com/browse/poly->. [Accessed: 14-Mar-2018].
- [8] "Morpheus, the God of Dreams Morpheus." [Online]. Available: <http://www.greekmyths-greekmythology.com/morpheus-the-god-of-dreams/>. [Accessed: 14-Mar-2018].
- [9] "Javanotes 7.0, Section 5.5 -- Inheritance, Polymorphism, and Abstract Classes." [Online]. Available: <http://math.hws.edu/javanotes/c5/s5.html>. [Accessed: 13-Mar-2018].
- [10] "Polymorphism (C# Programming Guide) | Microsoft Docs." [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/polymorphism>. [Accessed: 13-Mar-2018].
- [11] B. Stroustrup, "Stroustrup: C++ Glossary." [Online]. Available: <http://www.stroustrup.com/glossary.html#Gpolymorphism>. [Accessed: 23-Mar-2018].
- [12] "Nullable Types (C# Programming Guide) | Microsoft Docs." [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/nullable-types/>. [Accessed: 15-Mar-2018].
- [13] S. Kendal, *Object Oriented Programming Using C# .NET*. Simon Kendal & Ventus Publishing ApS, 2011.
- [14] D. Clark, "Beginning C # Object Oriented Programming, 2nd Ed.," p. 373, 2013.
- [15] S. Nakov, D. Dimitrov, H. Germanov, I. Murdanliev, M. Stoyanov, and M. Valkov, *Fundamentals of Computer Programming with C# (The Bulgarian C# Programming Book)*. 2013.
- [16] J. Sonmez, "Basic to Basics: What is Dependency Inversion? Is it IoC? Part 1 - Simple Programmer." [Online]. Available: <https://simpleprogrammer.com/basic-to-basics-what-is-dependency-inversion-is-it-ioc-part-1/>. [Accessed: 17-Mar-2018].
- [17] R. C. Martin, "The Dependency Inversion Principle," *Depend. Invers. Princ.*, 1996.
- [18] "Dependency Inversion Principle | DevIQ." [Online]. Available: <http://deviq.com/dependency-inversion-principle/>. [Accessed: 17-Mar-2018].
- [19] B. C. Martin Robert, M. Micah Publisher, and P. Hall, *Agile Principles, Patterns, and Practices in C#*. 2006.
- [20] D. Thomas and A. Hunt, *The Pragmatic Programmer*, 1st ed. Addison Wesley, 1999.
- [21] "Examining the Dependency Inversion Principle · Los Techies." [Online]. Available: <https://lostechies.com/derekgreer/2008/12/28/examining-dependency-inversion/>.

- [Accessed: 23-Mar-2018].
- [22] "Dependency Inversion Principle | Object Oriented Design." [Online]. Available: <http://www.oodesign.com/dependency-inversion-principle.html>. [Accessed: 23-Mar-2018].
 - [23] "Dependency Inversion Principle." [Online]. Available: https://www.eventhelix.com/RealtimeMantra/Object_Oriented/dependency_inversion_principle.htm#.WrTtvojFKUk. [Accessed: 23-Mar-2018].
 - [24] "Dependency Inversion Principle." [Online]. Available: <https://www.c-sharpcorner.com/UploadFile/pranayamr/dependency-inversion-principle/>. [Accessed: 19-Mar-2018].
 - [25] B. Bates, K. Sierra, E. Freeman, and E. Robson, *Head First Design Patterns*. O'Reilly, 2009.
 - [26] R. Zafar, "What is Software Testing?," *Code Project*. [Online]. Available: <http://istqbexamcertification.com/what-is-software-testing/>. [Accessed: 21-Mar-2018].
 - [27] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. New Jersey: John Wiley & Sons, 2012.
 - [28] C. Armitage, "Britain's biggest IT disasters in 2014 - Telegraph," *Telegraph*. [Online]. Available: <https://www.telegraph.co.uk/technology/news/11307560/Britains-biggest-IT-disasters-in-2014.html>. [Accessed: 21-Mar-2018].
 - [29] D. Hyde, "Five million UK workers face uncertainty after tax bills wrongly calculated twice in HMRC blunder - Telegraph," *Telegraph*. [Online]. Available: <https://www.telegraph.co.uk/finance/personalfinance/tax/11150266/Five-million-UK-workers-face-uncertainty-after-tax-bills-wrongly-calculated-twice-in-HMRC-blunder.html>. [Accessed: 21-Mar-2018].
 - [30] R. Neate, "Amazon sellers hit by nightmare before Christmas as glitch cuts prices to 1p | Technology | The Guardian," *The Guardian*. [Online]. Available: <https://www.theguardian.com/money/2014/dec/14/amazon-glitch-prices-penny-repricerexpress>. [Accessed: 21-Mar-2018].
 - [31] W. Ashford, "NHS to scrap £356m outpatient booking system," *Computer Weekly*. [Online]. Available: <http://www.computerweekly.com/news/2240220437/NHS-to-scrap-356m-outpatient-booking-system>. [Accessed: 21-Mar-2018].
 - [32] "Types of Software Testing - Complete List." [Online]. Available: <https://www.testingexcellence.com/types-of-software-testing-complete-list/>. [Accessed: 21-Mar-2018].
 - [33] "What is software testing? What are the different types of testing? - CodeProject." [Online]. Available: <https://www.codeproject.com/Tips/351122/What-is-software-testing-What-are-the-different-ty>. [Accessed: 21-Mar-2018].
 - [34] "Black Box Testing - Software Testing Fundamentals." [Online]. Available: <http://softwaretestingfundamentals.com/black-box-testing/>. [Accessed: 21-Mar-2018].
 - [35] "What is black-box, Specification-based, also known as behavioral testing techniques?" [Online]. Available: <http://istqbexamcertification.com/what-is-black-box-specification-based-also-known-as-behavioral-testing-techniques/#more-352>. [Accessed: 21-Mar-2018].
 - [36] "Boundary Testing." [Online]. Available: https://www.tutorialspoint.com/software_testing_dictionary/boundary_testing.htm. [Accessed: 23-Mar-2018].
 - [37] "White Box Testing - Software Testing Fundamentals." [Online]. Available: <http://softwaretestingfundamentals.com/white-box-testing/>. [Accessed: 21-Mar-2018].

- [38] "What is White Box Testing? Webopedia Definition." [Online]. Available: https://www.webopedia.com/TERM/W/White_Box_Testing.html. [Accessed: 22-Mar-2018].
- [39] "What is white-box or Structure-based or structural testing techniques?" [Online]. Available: <http://istqbexamcertification.com/what-is-white-box-or-structure-based-or-structural-testing-techniques/>. [Accessed: 22-Mar-2018].
- [40] "The Beginner's Guide to Unit Testing: What Is Unit Testing?" [Online]. Available: <https://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728>. [Accessed: 22-Mar-2018].
- [41] "Unit Testing - Software Testing Fundamentals." [Online]. Available: <http://softwaretestingfundamentals.com/unit-testing/>. [Accessed: 22-Mar-2018].
- [42] "List(T) Class (System.Collections.Generic)." [Online]. Available: [https://msdn.microsoft.com/en-us/library/6sh2ey19\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx). [Accessed: 23-Mar-2018].
- [43] "Unit Test Basics in Visual Studio - Visual Studio | Microsoft Docs." [Online]. Available: <https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics>. [Accessed: 23-Mar-2018].

Bibliography

- "c# - Formatting a double to two decimal places - Stack Overflow." [Online]. Available: <https://stackoverflow.com/questions/18418668/formatting-a-double-to-two-decimal-places>. [Accessed: 23-Mar-2018].
- "What is BLACK Box Testing? Techniques, Example & Types." [Online]. Available: <https://www.guru99.com/black-box-testing.html>. [Accessed: 22-Mar-2018].
- "UML, Abstract Classes and Methods, and Interfaces | The Oxford Math Center." [Online]. Available: <http://www.oxfordmathcenter.com/drupal7/node/35>. [Accessed: 22-Mar-2018].
- "UML 2 Class Diagram Guidelines." [Online]. Available: <http://www.agilemodeling.com/style/classDiagram.htm>. [Accessed: 22-Mar-2018].
- "UML Class Diagrams." [Online]. Available: <http://pages.cs.wisc.edu/~hasti/cs302/examples/UMLdiagram.html>. [Accessed: 22-Mar-2018].
- "Types of Software testing and definitions of testing terms — Software Testing Help." [Online]. Available: <http://www.softwaretestinghelp.com/types-of-software-testing/>. [Accessed: 21-Mar-2018].
- "Dependency Inversion Principle in C# – SOLID Design Principles – Part 5 « John Nelson's Blog." [Online]. Available: <https://johnnelson.com/2014/07/23/dependency-inversion-principle-in-c-solid-design-principles/>. [Accessed: 19-Mar-2018].
- "Dependency Inversion Principle | DevIQ." [Online]. Available: <http://deviq.com/dependency-inversion-principle/>. [Accessed: 19-Mar-2018].
- "Dependency Inversion Principle | DevIQ." [Online]. Available: <http://deviq.com/dependency-inversion-principle/>. [Accessed: 17-Mar-2018].
- "Inherit Multiple Interfaces With the Same Method Name in C#." [Online]. Available: <https://www.c-sharpcorner.com/UploadFile/rohatash/inherit-multiple-interfaces-with-the-same-method-name-in-C-Sharp/>. [Accessed: 17-Mar-2018].
- "Lesson 13: Interfaces – C# Station." [Online]. Available: <http://csharp-station.com/Tutorial/CSharp/Lesson13>. [Accessed: 17-Mar-2018].
- "Lesson 9: Polymorphism – C# Station." [Online]. Available: <http://csharp-station.com/Tutorial/CSharp/Lesson09>. [Accessed: 13-Mar-2018].
- "Programming - Object Oriented Programmin." [Online]. Available: <https://www.cs.utah.edu/~germain/PPS/Topics/oop.html>. [Accessed: 13-Mar-2018].
- C. Shokoples, *The Black Art of Piton Craft*, no. c. 2003.

M. Yaiser, "OOP concepts: Polymorphism and interfaces | Adobe Developer Connection," 2012. [Online]. Available: <https://www.adobe.com/devnet/actionscript/learning/oop-concepts/polymorphism-and-interfaces.html>. [Accessed: 13-Mar-2018].

Appendix A

Exercise One

Program

```
namespace ExerciseOne
{
    class Program
    {
        static void Main(string[] args)
        {
            // Set value of spoonTotal to 0 and set value of spoonCost to 0.20p.
            Counter CountSpoons = new Counter(0, 0.20);

            // Manually set amount of spoons to test program.
            //CountSpoons.SetSpoons(50);

            // Allow user to set amount of spoons to test program.
            Console.WriteLine(" -----");

            // Prompt user to enter spoons manufactured
            Console.Write(" Enter Total Spoons Manufactured by Machine: ");

            // Read amount of spoons entered and add to count of spoons set to 0 upon
            // creation of spoon counter.
            int machineTotal = int.Parse(Console.ReadLine());
            CountSpoons.SetSpoons(CountSpoons.GetSpoonTotal() + machineTotal);

            Console.WriteLine("");
            // Display total amount of spoons entered.
            Console.WriteLine(" Total Spoons Manufactured: {0} \n",
CountSpoons.GetSpoonTotal());

            // Assign TotalSpoonCost method's returned value into variable
            // 'totalSpoonCost' to pass to string formatter method CostFormatter.
            double totalSpoonCost = CountSpoons.TotalSpoonCost();
            Console.WriteLine(" Total Manufactured Cost of Spoons: £{0}p",
CountSpoons.CostFormatter(totalSpoonCost));
            Console.WriteLine(" -----");

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

Counter Class

```
namespace ExerciseOne
{
    class Counter
    {
        // Attributes

        private int _totalSpoons;
        private double _spoonCost;

        // Constructors
        public Counter(int totalSpoons, double spoonCost)
        {
            _totalSpoons = totalSpoons;
            _spoonCost = spoonCost;
        }

        // Properties
        public void SetSpoons(int spoonTotal)
        {
            _totalSpoons = spoonTotal;
        }
        public int GetSpoonTotal()
        {
            return _totalSpoons;
        }

        // Methods
        public double TotalSpoonCost()
        {
            return (_totalSpoons * _spoonCost);
        }
        public string CostFormatter(double formattedTotal)
        {
            string formattedTotalCost = String.Format("{0:0.00}", formattedTotal);
            return formattedTotalCost;
        }
    }
}
```

Exercise Two

Program

```
namespace ExerciseTwo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Prompt user to enter Circle One's Values
            Console.WriteLine(" -----");
            Console.Write(" Enter Circle One X Value: ");
            // Read Value X and store in variable pointX.
            int pointX1 = int.Parse(Console.ReadLine());
            // Enter and Read Value Y and store in variable pointY
            Console.Write(" Enter Circle One Y Value: ");
            int pointY1 = int.Parse(Console.ReadLine());
            // Enter and Read Radius Value
            Console.Write(" Enter Circle One Radius: ");
            double radius1 = double.Parse(Console.ReadLine());
            Console.WriteLine(" -----");

            // Prompt user to enter Circle Two's values
            Console.Write(" Enter Circle Two X Value: ");
            // Read Value X and store in variable pointX.
            int pointX2 = int.Parse(Console.ReadLine());
            // Enter and Read Value Y and store in variable pointY
            Console.Write(" Enter Circle Two Y Value: ");
            int pointY2 = int.Parse(Console.ReadLine());
            //Enter and Read Radius Value
            Console.Write(" Enter Circle Two Radius: ");
            double radius2 = double.Parse(Console.ReadLine());
            Console.WriteLine(" ----- \n");

            // Set value of Circle1 x, y, r passing in values user has entered
            Circle C1 = new Circle(pointX1, pointY1, radius1);
            // Set value of Circle2 x, y, r
            Circle C2 = new Circle(pointX2, pointY2, radius2);

            if (C1.CheckOverlap(C2) == true)
            {
                Console.WriteLine(" CIRCLES OVERLAPPING!");
            }
            else
            {
                Console.WriteLine(" CIRCLES NOT OVERLAPPING!");
            }

            //Console.WriteLine(C1.CheckHypotenuse(C2));

            //Console.WriteLine(C1.CheckOverlap(C2));

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

Circle Class

```

namespace ExerciseTwo
{
    class Circle
    {
        // Attributes
        private int _x;
        private int _y;
        private double _r;

        // Constructors
        public Circle(int x, int y, double r)
        {
            _x = x;
            _y = y;
            _r = r;
        }

        // Properties
        public int GetPointX()
        {
            return _x;
        }
        public int GetPointY()
        {
            return _y;
        }
        public double GetRadius()
        {
            return _r;
        }

        // Methods
        public bool CheckOverlap(Circle circle) // Pass in existing Circle object.
        {
            int a = _x - circle.GetPointX(); // Get in properties is acting as second
            int b = _y - circle.GetPointY();
            double sumRadii = _r + circle.GetRadius();
            double hypotenuse = Math.Sqrt(a * a + b * b);

            if (sumRadii > hypotenuse)
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        // Hypotenuse Test
        public double CheckHypotenuse(Circle circle) // Pass in existing Circle
        object.
        {
            int a = _x - circle.GetPointX();
            int b = _y - circle.GetPointY();
            double sumRadii = _r + circle.GetRadius();
            double hypotenuse = Math.Sqrt(a * a + b * b);
            return hypotenuse;
        }
    }
}

```

Exercise Three

Program

```
namespace Exercise3
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create cart.
            Cart Cart = new Cart();
            // Create Book.
            Book Book = new Book("C# for Dummies", 11.99);
            // Add book to cart.
            Cart.AddProduct(Book);

            // Create DVD.
            DVD DVD = new DVD("The Deer Hunter", 8.99);
            // Add DVD to cart.
            Cart.AddProduct(DVD);

            // Create CD/
            CD CD = new CD("Britney Spears", 3.99);
            // Add CD to cart
            Cart.AddProduct(CD);

            //Console.WriteLine(DVD.GetProductTax());
            Console.WriteLine(" ----- CART CONTENTS ----- ");
            Console.WriteLine(" {0} £{1}", Book.ProductName(),
            Book.GetProductPrice());
            Console.WriteLine(" {0} £{1}", DVD.ProductName(), DVD.GetProductPrice());
            Console.WriteLine(" {0} £{1}", CD.ProductName(), CD.GetProductPrice());
            Console.WriteLine(" -----");
            Console.WriteLine(" Sub Total: £{0}", Cart.SubTotalProductCost());
            Console.WriteLine(" Total Tax: {0}%", Cart.TotalProductTax());
            Console.WriteLine(" Total Tax: £" + string.Format("{0:0.00}",
            Cart.TotalProductTaxCost())); // Format to two decimal places.
            Console.WriteLine(" Grand Total: £" + string.Format("{0:0.00}",
            Cart.TotalProductCost()));
            Console.WriteLine(" -----");
            //Console.WriteLine("Earnings this week: {0:0.00}", answer);
            //Console.WriteLine("Earnings this week: " + string.Format("{0:0.00}",
            answer));
            // Possibly add in list of products and assign each product to a variable
            // to allow user to select products to add.
            Console.ReadKey();
        }
    }
}
```

IProduct Interface

```
namespace Exercise3
{
    interface IProduct
    {
        // Product Name Method
        string ProductName();
        // Product Price Method
        double GetProductPrice();
        // Product Tax Method
        double GetProductTax();
    }
}
```

Cart Class – On next page.

```

namespace Exercise3
{
    class Cart
    {
        // Attributes
        // Create list to add products to in the shopping cart.
        private List<IProduct> _cartContents = new List<IProduct>();

        // Constructors

        // Properties

        // Methods
        // Add products to list of products in the shopping cart.
        public void AddProduct(IProduct item)
        {
            _cartContents.Add(item);
        }

        public double SubTotalProductCost()
        {
            // Set total of cart to 0. // Outside foreach to avoid setting back to 0
            // everytime.
            double subTotal = 0;
            // Loop through each product in cart and add cost if items before tax.
            foreach (IProduct product in _cartContents)
            {
                subTotal = subTotal + product.GetProductPrice();
            }
            //return grandTotal;
            return subTotal;
        }

        public double TotalProductTax()
        {
            double tax = 0;
            // Loop through each product in cart and add total tax percentage to be added
            // to cart cost.
            foreach (IProduct product in _cartContents)
            {
                tax = tax + product.GetProductTax();
            }
            return tax;
        }

        public double TotalProductTaxCost()
        {
            double subtotal = 0;
            double tax = 0;
            double totalTax = 0;
            // Loop through each product in cart and work out total cost of tax on items
            // in cart.
            foreach (IProduct product in _cartContents)
            {
                subtotal = subtotal + product.GetProductPrice();
                tax = tax + product.GetProductTax();
                totalTax = subtotal * tax / 100;
            }
            //return grandTotal;
            return totalTax;
        }
    }
}

```

DVD Class

```
namespace Exercise3
{
    class DVD : IProduct
    {
        // Attributes

        private string _title;
        private double _price;

        // Constructors
        public DVD(string title, double price)
        {
            _title = title;
            _price = price;
        }

        // Properties
        public double GetProductPrice()
        {
            return _price;
        }

        public double GetProductTax()
        {
            return 12;
        }

        public string ProductName()
        {
            return _title;
        }

        // Methods
    }
}
```

CD Class

```
namespace Exercise3
{
    class CD : IProduct
    {
        // Attributes

        private string _title;
        private double _price;

        // Constructors
        public CD(string title, double price)
        {
            _title = title;
            _price = price;
        }
        // Properties
        public double GetProductPrice()
        {
            return _price;
        }

        public double GetProductTax()
        {
            return 10;
        }

        public string ProductName()
        {
            return _title;
        }

        // Methods
    }
}
```

Book Class

```
namespace Exercise3
{
    class Book : IProduct
    {
        // Attributes

        private string _title;
        private double _price;

        // Constructors
        public Book(string title, double price)
        {
            _title = title;
            _price = price;
        }

        // Properties
        public double GetProductPrice()
        {
            return _price;
        }

        public double GetProductTax()
        {
            return 0;
        }

        public string ProductName()
        {
            return _title;
        }

        // Methods
    }
}
```

Appendix B

Test Program Code

Class & Object Example

```
class Program
{
    static void Main(string[] args)
    {
        /* MEDIUM SIZE CAR OBJECT */

        Car Medium = new Car();

        Medium.SetManufacturer("Volkswagen");
        Medium.SetModel("Golf");
        Medium.SetEngineSize(2000);
        Medium.SetFuelType("Diesel");
        Medium.SetColour("Black");

        Console.WriteLine("| MANUFACTURER | MODEL | ENGINE SIZE | FUEL TYPE |
COLOUR |");
        Console.WriteLine("| {0} | {1} | {2} | {3} | {4} |",
Medium.GetManufacturer(),
Medium.GetModel(), Medium.GetEngineSize(), Medium.GetFuelType(),
Medium.GetColour());

        /* SMALL SIZE CAR OBJECT */

        Car Small = new Car();

        Small.SetManufacturer("Renault");
        Small.SetModel("Clio");
        Small.SetEngineSize(1200);
        Small.SetFuelType("Petrol");
        Small.SetColour("Red");

        Console.WriteLine("| {0} | {1} | {2} | {3} | {4} |",
Small.GetManufacturer(),
Small.GetModel(), Small.GetEngineSize(), Small.GetFuelType(),
Small.GetColour());

        /* MICRO SIZE CAR OBJECT */

        Car Micro = new Car();

        Micro.SetManufacturer("Citroen");
        Micro.SetModel("C2");
        Micro.SetEngineSize(998);
        Micro.SetFuelType("Petrol");
        Micro.SetColour("Pink");

        Console.WriteLine("| {0} | {1} | {2} | {3} | {4} |",
Micro.GetManufacturer(),
Micro.GetModel(), Micro.GetEngineSize(), Micro.GetFuelType(),
Micro.GetColour());

        /* FAMILY SIZE CAR OBJECT */

        Car Family = new Car();

        Family.SetManufacturer("Vauxhall");
        Family.SetModel("Zafira");
        Family.SetEngineSize(1600);
        Family.SetFuelType("Diesel");
        Family.SetColour("Grey");
```

```

namespace Class_Object_Example_ASS
{
    class Car
    {
        // Attributes

        private string _manufacturer;
        private string _model;
        private int _engineSize;
        private string _fuelType;
        private string _colour;

        // Constructors

        // Properties

        /* SET VALUES */
        public void SetManufacturer(string value)
        {
            _manufacturer = value;
        }
        public void SetModel(string value)
        {
            _model = value;
        }
        public void SetEngineSize(int value)
        {
            _engineSize = value;
        }
        public void SetFuelType(string value)
        {
            _fuelType = value;
        }
        public void SetColour(string value)
        {
            _colour = value;
        }

        /* GET VALUES */

        public string GetManufacturer()
        {
            return _manufacturer;
        }
        public string GetModel()
        {
            return _model;
        }
        public int GetEngineSize()
        {
            return _engineSize;
        }
        public string GetFuelType()
        {
            return _fuelType;
        }
        public string GetColour()
        {
            return _colour;
        }

        // Methods
    }
}

```

Inheritance

```
namespace Inheritance
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(" VEHICLE CLASS");
            Vehicle vehicle = new Vehicle();
            vehicle.PrintManufacturer();
            vehicle.PrintYear();

            Console.WriteLine("-----");

            Console.WriteLine(" CAR CLASS");
            Car car = new Car();
            car.PrintManufacturer();
            car.PrintYear();

            Console.ReadKey();
        }
    }
}

namespace Inheritance
{
    class Vehicle
    {
        // Attributes
        // Constructors
        // Properties
        // Methods
        public void PrintManufacturer()
        {
            Console.WriteLine(" Who made me?");
        }
        public void PrintYear()
        {
            Console.WriteLine(" When was I made?");
        }
    }
}

namespace Inheritance
{
    class Car : Vehicle
    {
        // Attributes

        // Constructors

        // Properties

        // Methods
    }
}
```

Polymorphism

```
namespace Polymorphism_Example
{
    class Program
    {
        static void Main(string[] args)
        {
            //Vehicle vehicle = new Vehicle();
            //Console.WriteLine("-- MANUFACTURED VEHICLE METHOD --");
            //Console.WriteLine(vehicle.ManufactureVehicle());
            //Console.WriteLine("-- YEAR METHOD --");
            //Console.WriteLine(vehicle.Year());
            //Console.WriteLine("-- NO OF WHEELS METHOD --");
            //Console.WriteLine(vehicle.NoOfWheels());

            List<Vehicle> vehicles = new List<Vehicle>();

            vehicles.Add(new Plane());
            vehicles.Add(new Ship());
            vehicles.Add(new Car());

            foreach (Vehicle manufactured in vehicles)
            {
                Console.WriteLine(" ----- ");
                Console.WriteLine(" I am a {0}, I was made in {1} and I have {2}
                wheels \n",
                    manufactured.ManufactureVehicle(), manufactured.Year(),
                    manufactured.NoOfWheels());
                Console.WriteLine(" ----- ");
            }

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

```

namespace Polymorphism_Example
{
    class Vehicle
    {
        // Attributes
        // Constructors
        // Properties
        // Methods
        public virtual string ManufactureVehicle()
        {
            return "I'm just a collection of metal, nuts, bolts and parts, what would
you like me to be? \n";
        }
        public virtual int? NoOfWheels() // Set to nullable int to allow null value.
        {
            return null;
        }
        public virtual string Year()
        {
            return "What year was I made? \n";
        }
    }
}

namespace Polymorphism_Example
{
    class Ship : Vehicle
    {
        // Attributes
        // Constructors
        // Properties
        // Methods
        public override string ManufactureVehicle()
        {
            return "SHIP!";
        }
        public override int? NoOfWheels()
        {
            return 0;
        }
        public override string Year()
        {
            return "1991";
        }
    }
}

```

```

namespace Polymorphism_Example
{
    class Plane : Vehicle
    {
        // Attributes
        // Constructors
        // Properties
        // Methods
        public override string ManufactureVehicle()
        {
            return "PLANE!";
        }
        public override int? NoOfWheels()
        {
            return 18;
        }
        public override string Year()
        {
            return "2005";
        }
    }
}

namespace Polymorphism_Example
{
    class Car : Vehicle
    {
        // Attributes
        // Constructors
        // Properties
        // Methods
        public override string ManufactureVehicle()
        {
            return "CAR!";
        }

        public override int? NoOfWheels()
        {
            return 4;
        }

        public override string Year()
        {
            return "2018";
        }
    }
}

```

Interfaces Example

```
namespace Interfaces_Ex
{
    class Program
    {
        static void Main(string[] args)
        {
            VehicleFactory factory = new VehicleFactory();
            ICar SportsCar = factory.CreateSportsCar();
            ICar FamilyCar = factory.CreateFamilyCar();
            ICar CityCar = factory.CreateCityCar();
            IEngine SportsCarEngine = factory.CreateSportsCarEngine();
            IEngine FamilyCarEngine = factory.CreateFamilyCarEngine();
            IEngine CityCarEngine = factory.CreateCityCarEngine();

            Console.WriteLine(" -----");
            Console.WriteLine(" ---- SPORTS CAR ----");
            Console.WriteLine(" Manufacturer: {0}", SportsCar.GetManufacturerName());
            Console.WriteLine(" Model: {0}", SportsCar.GetModelName());
            Console.WriteLine(" Engine Size: {0}", SportsCarEngine.GetEngineSize());
            Console.WriteLine(" Engine Size: {0}", SportsCarEngine.GetEngineType());
            Console.WriteLine(" Engine Size: {0}", SportsCarEngine.GetFuelType());
            Console.WriteLine(" Seats: {0}", SportsCar.GetNoOfSeats());
            Console.WriteLine(" Doors: {0}", SportsCar.GetNoOfDoors());
            Console.WriteLine(" -----");
            Console.WriteLine(" ---- FAMILY CAR ----");
            Console.WriteLine(" Manufacturer: {0}", FamilyCar.GetManufacturerName());
            Console.WriteLine(" Model: {0}", FamilyCar.GetModelName());
            Console.WriteLine(" Engine Size: {0}", FamilyCarEngine.GetEngineSize());
            Console.WriteLine(" Engine Size: {0}", FamilyCarEngine.GetEngineType());
            Console.WriteLine(" Engine Size: {0}", FamilyCarEngine.GetFuelType());
            Console.WriteLine(" Seats: {0}", FamilyCar.GetNoOfSeats());
            Console.WriteLine(" Doors: {0}", FamilyCar.GetNoOfDoors());
            Console.WriteLine(" -----");
            Console.WriteLine(" ---- CITY CAR ----");
            Console.WriteLine(" Manufacturer: {0}", CityCar.GetManufacturerName());
            Console.WriteLine(" Model: {0}", CityCar.GetModelName());
            Console.WriteLine(" Engine Size: {0}", CityCarEngine.GetEngineSize());
            Console.WriteLine(" Engine Size: {0}", CityCarEngine.GetEngineType());
            Console.WriteLine(" Engine Size: {0}", CityCarEngine.GetFuelType());
            Console.WriteLine(" Seats: {0}", CityCar.GetNoOfSeats());
            Console.WriteLine(" Doors: {0}", CityCar.GetNoOfDoors());
            Console.WriteLine(" -----");

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

```

namespace Interfaces_Ex
{
    interface ICar
    {
        // String Method for Car's Manufacturer
        string GetManufacturerName();

        // String Method for Car's Model Name
        string GetModelName();

        // Int Method for Car's Number of Seats
        int GetNoOfSeats();

        // Int Method for Car's Number of Doors
        int GetNoOfDoors();
    }
}

namespace Interfaces_Ex
{
    interface IEngine
    {
        // String Method for Engine Size
        string GetEngineSize();

        // String Method for Engine Type
        string GetEngineType();

        // String Method for Engine Fuel Type
        string GetFuelType();
    }
}

```

```

namespace Interfaces_Ex
{
    class VehicleFactory
    {
        public ICar CreateSportsCar()
        {
            SportsCar SportsCar = new SportsCar();
            return SportsCar;
        }
        public ICar CreateFamilyCar()
        {
            FamilyCar FamilyCar = new FamilyCar();
            return FamilyCar;
        }
        public ICar CreateCityCar()
        {
            CityCar CityCar = new CityCar();
            return CityCar;
        }
        public IEngine CreateSportsCarEngine()
        {
            SportsCar SportsCarEngine = new SportsCar();
            return SportsCarEngine;
        }
        public IEngine CreateFamilyCarEngine()
        {
            FamilyCar FamilyCarEngine = new FamilyCar();
            return FamilyCarEngine;
        }
        public IEngine CreateCityCarEngine()
        {
            CityCar CityCarEngine = new CityCar();
            return CityCarEngine;
        }
    }
}

```

```

namespace Interfaces_Ex
{
    class CityCar : ICar, IEngine
    {
        // String Method for Car's Manufacturer
        public string GetManufacturerName()
        {
            return "Mercedes Smart";
        }

        // String Method for Car's Model Name
        public string GetModelName()
        {
            return "ForTwo";
        }

        // Int Method for Car's Number of Seats
        public int GetNoOfSeats()
        {
            return 2;
        }

        // Int Method for Car's Number of Doors
        public int GetNoOfDoors()
        {
            return 2;
        }

        // String Method for Car's Engine // From IEngine
        public string GetEngineSize()
        {
            return "800cc";
        }

        // String Method for Engine Type // From IEngine
        public string GetEngineType()
        {
            return "Inline Three Cylinder";
        }

        // String Method for Engine Fuel Type // From IEngine
        public string GetFuelType()
        {
            return "Petrol";
        }
    }
}

```

```

namespace Interfaces_Ex
{
    class FamilyCar : ICar, IEngine
    {
        // String Method for Car's Manufacturer
        public string GetManufacturerName()
        {
            return "Volkswagen";
        }
        // String Method for Car's Model Name
        public string GetModelName()
        {
            return "Touran";
        }
        // Int Method for Car's Number of Seats
        public int GetNoOfSeats()
        {
            return 7;
        }
        // Int Method for Car's Number of Doors
        public int GetNoOfDoors()
        {
            return 5;
        }
        // String Method for Car's Engine // From IEngine
        public string GetEngineSize()
        {
            return "1998cc";
        }
        // String Method for Engine Type // From IEngine
        public string GetEngineType()
        {
            return "Inline four Cylinder";
        }
        // String Method for Engine Fuel Type // From IEngine
        public string GetFuelType()
        {
            return "Diesel";
        }
    }
}

```

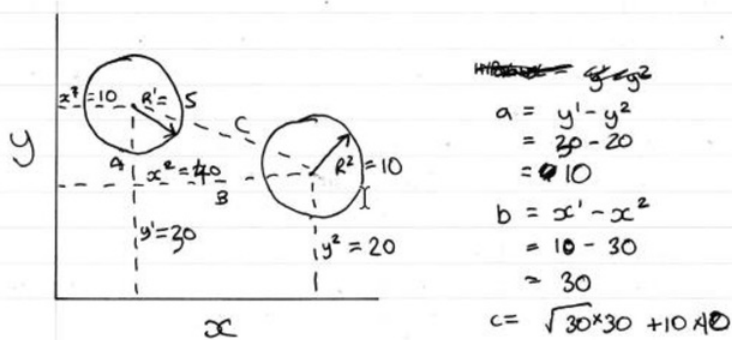
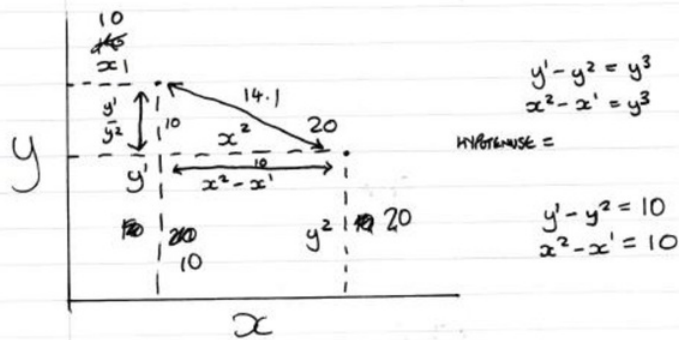
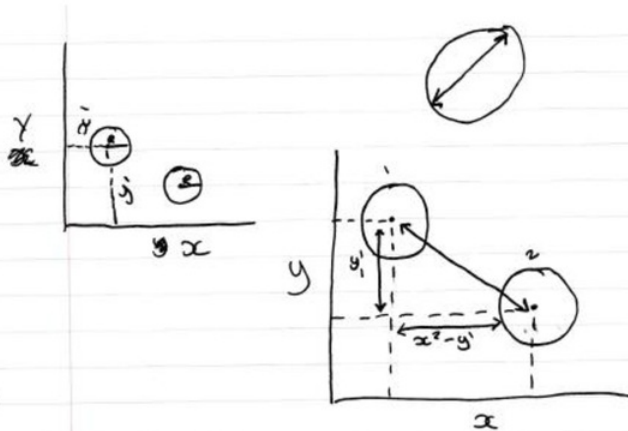
```

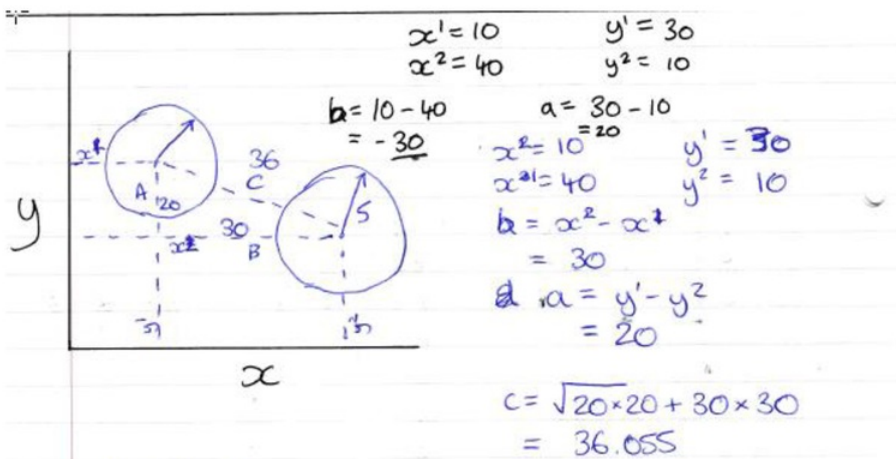
namespace Interfaces_Ex
{
    class SportsCar : ICar, IEngine
    {
        // Attributes
        // Constructors
        // Properties
        // Methods

        // String Method for Car's Manufacturer
        public string GetManufacturerName()
        {
            return "Porsche";
        }
        // String Method for Car's Model Name
        public string GetModelName()
        {
            return "911";
        }
        // Int Method for Car's Number of Seats
        public int GetNoOfSeats()
        {
            return 2;
        }
        // Int Method for Car's Number of Doors
        public int GetNoOfDoors()
        {
            return 2;
        }
        // String Method for Car's Engine // From IEngine
        public string GetEngineSize()
        {
            return "2998cc";
        }
        // String Method for Engine Type // From IEngine
        public string GetEngineType()
        {
            return "Boxer Engine";
        }
        // String Method for Engine Fuel Type // From IEngine
        public string GetFuelType()
        {
            return "Petrol";
        }
    }
}

```

Appendix C



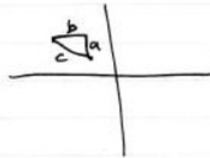


RADIUS1 = 5
 RADIUS2 = 5
 TOTAL = 10

$C1 = 40, 30, 5$ $C2 = 10, 10, 5$

Ex 3

CART
~~- CART~~



EXAMPLE COORDINATES = $C1 = 40, 30, 5$ $C2 = 10, 10, 5$
 $C1 = 40, 30, 20$ $C2 = 10, 10, 20$

TEST 1

TEST 2

$C1 = 20, 30, 10.5$

$C1 = 20, 30, 20.2$

$C2 = 40, 20, 10.5$

$C2 = 40, 20, 20.4$

FINAL GRADE

72/100

GENERAL COMMENTS

Instructor

Knowledge OOPs concepts (30) 26

Discussion of design principles (10) 8

Knowledge of testing (10) 8

UML (20) 15

Exercise 1 (5) 4

Exercise 2 (5) 3

Exercise 3 (10) 8

Exercise 4 (10) 0

Classes vs Objects: just a little confusion at the start of this description

Inheritance and Polymorphism: very detailed.

Interfaces: again very detailed explanation

Dependency Inversion: interesting discussion, again good level of detail provided

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

UML: Good attempt, a couple of errors in the class diagram. You should have provided a couple of sequence diagrams for the main scenarios

Ex1: A method was required to count a spoon being manufactured, whereas you provided a method for just changing the amount of spoons

Ex2: Did the calculations but you have created the overlap test.

Ex3: Calculation are over complicated

PAGE 1

PAGE 2

PAGE 3

PAGE 4



Comment 1

I think this is the class.

Perhaps what you mean is an object is a specific instance of a car



Comment 2

Not sure what this is referring to.



Comment 3

I think you would have been to refer to it "containing the specific values"

We talk about attributes when describing the names and types of the values in the class.



Comment 4

This is more clear

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10



Comment 5

This is the important point, we can use the abstract nature of inheritance and polymorphism to allow us to subdivide our code, and apply the Open Closed Principle



Comment 6

Good, this is the important point

You could have referred to interfaces defining "roles" that classes can play



Comment 7

Interesting, because it also has a number of benefits when it comes to testing e.g. the ability to "inject" test objects because the dependency isn't being hidden by the class under test

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39



Comment 8

Not a problem and you asked permission

PAGE 40



Comment 9

Arrow head should have been drawn the other way around since Tester and Developer inherit from User



Comment 10

Actually the TestTeam class contains a list of ITester not Tester



Comment 11

The arrow heads should be reversed on the Use relationships e.g. the ProblemRepository uses the ProblemReport class



Comment 12

Admin implements ITester, so there is no direct relationship between Admin and TestTeam



Comment 13

See TestTeam notes

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

PAGE 57

PAGE 58

PAGE 59

PAGE 60

PAGE 61

PAGE 62

PAGE 63

PAGE 64

PAGE 65

PAGE 66

PAGE 67

PAGE 68

PAGE 69

PAGE 70

PAGE 71

