

1809693-MarkHuntly- AdvancedSoftwareDevAssignme ntReport

by Mark Huntly

FILE	11495_MARK_HUNTLY_1809693-MARKHUNTLY- ADVANCEDSOFTWAREDEVASSIGNMENTREPORT_226878_264092247.D OCX (317.69K)		
TIME SUBMITTED	17-FEB-2020 06:53PM (UTC+0000)	WORD COUNT	1827
SUBMISSION ID	120245983	CHARACTER COUNT	9560

This report explains the following: The difference between a class and an object, the term “data encapsulation”, interfaces and their impact on the process of developing code. Specifically, their role in the subdivision of code through abstraction. It will also show examples of code to illustrate this. There will also be tables for testing the author’s program for part two of the assignment.

HND/BSc Web Development

1809693 - Mark Huntly

Table of Contents

1.0	Introduction.....	2
2.0	Object Orientated Programming	2
2.1	Classes and an Objects	2
2.1.1	Classes.....	2
2.1.2	Objects	2
2.2	Data encapsulation	6
2.3	Interfaces.....	7
2.3.1	Abstraction.....	7
	References	10

1.0 Introduction

Within computer science, a program is a series of commands or algorithms running within a computer to process information and produce it for the user. Programming is the process of writing these algorithms. This can be split into two groups:

- **System Programming** - the low-level instructions used to handle and operate the computer hardware.
- **Application Programming** - a higher level programming process allowing the performance of specific tasks for the user.

This report will discuss one of the methods used to approach application programming called 'Object Orientated Programming' and the features available within this method. [1]

2.0 Object Orientated Programming

The popular programming paradigm, or method used within web development is known as Object Orientated Programming (OOP). OOP allows modular programming, making it easier to work on individual sections, and allowing quicker and easier development, fixes and execution. It is also able to allow simulation of real-world objects through building classes to create these objects and giving them attributes based on real-world items. [2]

2.1 Classes and an Objects

Objects are the foundation of OOP. The paradigm is formed on a collection of objects communicating with each other to perform the tasks required. These modular objects are what can simulate real world items. The way these objects are created is through the design of classes. This is commonly known as instantiating as you create an instance of the class.

2.1.1 Classes

The class is a data type, like a variable, to be defined. It is the design blueprint of an object. It holds the data and information needed to create an object but doesn't take any memory. Within the class will be the attributes and methods needed to create the physical object. It could be thought of as a logical prototype and will be defined by the programmer or end user.

2.1.2 Objects

Objects, more commonly known as instances, are the physical existence of the logic within the class. There is no limit to the number of objects created from a class, taking the state and behaviour from that class's attributes and methods, giving them values and storing them in memory. As an example of this, there could be a Car class created. Within Car would be the attributes of the car such as model, number of doors, colour and engine size. From this class instances, or objects of cars could be created, such as Ford, Vauxhall, Nissan and so on. These objects would hold values to the attributes in the class and would use memory. Furthermore, the class could have methods such as sounding the horn, accelerating or braking. This could be realised within the object as in the figure below.

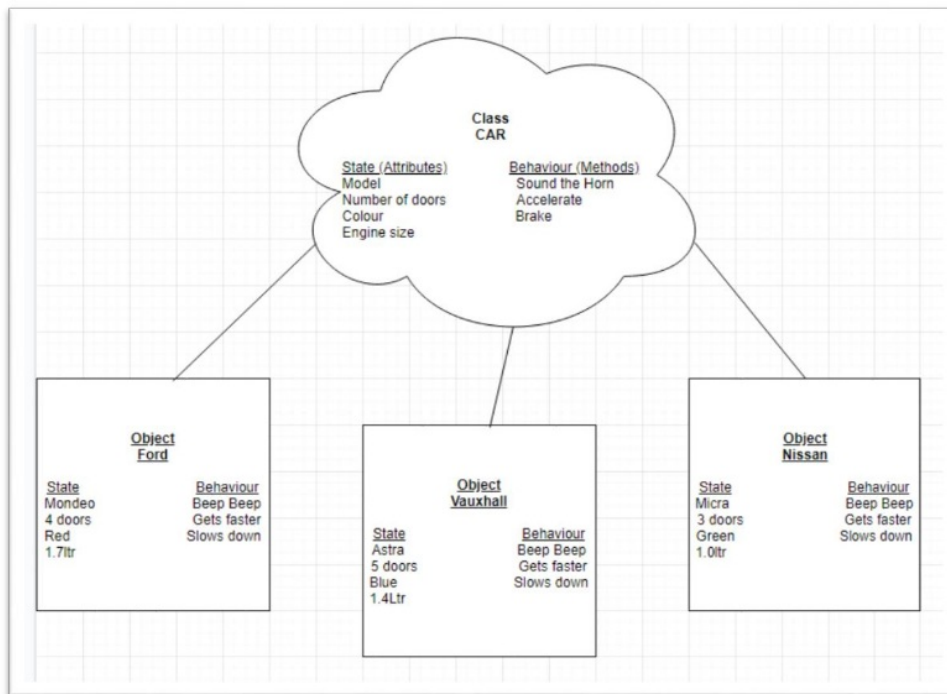


Figure 1- Illustration of a car object from a class

Consider the code for the example for a ford object could be written as follows.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Example
6  {
7      3 references
8      class Car
9      {
10         //Attributes
11         private string _model;
12         private string _numberOfDoors;
13         private string colour;
14         private string engineSize;
15
16         //Constructors
17         1 reference
18         public Car(string aModel, string aDoorNum, string aColour, string aEngineSize)
19         {
20             _model = aModel;
21             _numberOfDoors = aDoorNum;
22             colour = aColour;
23             engineSize = aEngineSize;
24         }
25
26         //Properties
27         1 reference
28         public string Model
  
```

Figure 2- Car class with constructor (state)

```

24
25 //Properties
26 1 reference
27 public string Model
28 {
29     get { return _model; }
30     set { _model = value; }
31 }
32 1 reference
33 public string DoorNum
34 {
35     get { return _numberOfDoors; }
36     set { _numberOfDoors = value; }
37 }
38 1 reference
39 public string Colour
40 {
41     get { return _colour; }
42     set { _colour = value; }
43 }
44 1 reference
45 public string EngineSize
46 {
47     get { return _engineSize; }
48     set { _engineSize = value; }
49 }
50
51 //Methods
52 1 reference
53 public void SoundHorn()

```

Figure 3 - Properties from the Car class

```

46
47 //Methods
48 1 reference
49 public void SoundHorn()
50 {
51     Console.WriteLine("Beep Beep");
52 }
53 1 reference
54 public void Accelerate()
55 {
56     Console.WriteLine("The {0} speeds up", _model);
57 }
58 1 reference
59 public void Brake()
60 {
61     Console.WriteLine("The {0} slows down", _model);
62 }
63 }
64
65

```

Figure 4 - Methods from the Car class (behaviour)

```

1  using System;
2
3  namespace Example
4  {
5      References
6      class Program
7      {
8          References
9          static void Main(string[] Args)
10         {
11             Car ford = new Car("Focus", "5 Door", "Red", "1.4Litres");
12             Console.WriteLine("The (0) (1) is a (2) and has a (3) engine.", ford.Colour, ford.Model, ford.DoorNum, ford.EngineSize);
13
14             ford.SoundHorn();
15             ford.Accelerate();
16             ford.Brake();
17         }
18     }
19 }
20
21
22

```

Figure 5 - Creating the Ford object in the main program

```

Microsoft Visual Studio Debug Console
The Red Focus is a 5 Door and has a 1.4Litres engine.
Beep Beep
The Focus speeds up
The Focus slows down

C:\Users\markh\source\repos\Example\Example\bin\Debug\netcore...
To automatically close the console when debugging stops, enable
the console when debugging stops.
Press any key to close this window . . .

```

Figure 6 - The output produced showing state and behaviour

To differentiate between a class and an object, the table below draws a concise comparison.

Class	Object
A class is a blueprint from which you can create the instance, i.e., objects.	An object is the instance of the class, which helps programmers to use variables and methods from inside the class.
A class is used to bind data as well as methods together as a single unit.	object acts as a variable of the class.
Classes have logical existence.	Objects have a physical existence.
A class doesn't take any memory spaces when a programmer creates one.	An object takes memory when a programmer creates one.
The class has to be declared only once.	Objects can be declared several times depending on the requirement.

Table 1 - Differences between a class and an object [3]

2.2 Data encapsulation

Data encapsulation is the process of enclosing or binding the attributes, data and methods inside a single unit (a class). The reasons for doing this is are:

- To hide the data and protect it from being manipulated or corrupted by people and to maintain its integrity.
- To abstract methods and data so that the user or caller only sees what they need to for their information.

A good analogy is the car. A driver will get in the car, turn the key or press the start button and the engine starts for them to drive. What goes into making that happen, the electronic connection, the starter motor turning, the spark to fire the fuel etc, is not seen by the driver, just the result of the process which is the engine starting. This is all that needs to be seen. By not being exposed, the parts of the car are protected from the driver, and the driver from adversely affecting the parts. Encapsulating the workings or hiding it within a class has the same affect. The way to implement encapsulation is to:

2

- Declare the attributes as private.
This restricts the access directly from outside the class and can only be accessed by methods and properties within the class.
- Allowing the population of the data through properties using getters and setters, and methods.
When getter and setter properties are defined, they access the attributes and allow the read and write functions to them. Methods can be written to populate the attributes too. This way the access can be tightly controlled with validation.
- Setting a public constructor to populate the attributes when an object is instantiated. As with the properties, a constructor can be controlled through proper validation of the data being input.

Once suitable and proper validation is set through the properties, it allows the correct data for the attributes to be applied. The correct and validated data can then be stored, as opposed to directly entering any data unvalidated straight into the attributes. [4] [5] [6] The example below demonstrates a the validation of the getters and setters for the 'DoorNum' property from the Car class to control the data entered into the '_numberOfDoors' attribute.

```
30 }
31 1reference
32 public string DoorNum
33 {
34     get { return _numberOfDoors; }
35     set
36     {
37         if (value == "2 door" || value == "3 door" || value == "4 door" || value == "5 door")
38         { _numberOfDoors = value; }
39         else
40         { Console.WriteLine("Please choose 2, 3, 4 or 5 door"); }
41     }
42 }
43 }
44 1reference
45 public string Colour
```

Figure 7 - A public property to control a private attribute

2.3 Interfaces

Interfaces are an integral part of modular OOP. They are often referred to as contracts because they will contain everything needed to perform the task from the class it implements. It could be looked on as an explicit contract between the calling code (eg the UI) and the called code (eg the class) allowing the de-coupling of said code. This provides some useful biproducts which include:

- Modular coding.
- Standardisation.
- Enforcement.
- Abstraction.

An interface is essentially a public template of a class, bearing the signatures of the methods and properties needed from it. It does not necessarily need all the properties and methods within the class and could be a partial template which would help the abstraction process. It can be used as a data type within the program code that will access the contract specification within the interface, no more and no less. Either side of the interface, in this case the UI and class, will have no knowledge of the other or what they contain or do. They will only know the details of the contract within the interface and will implement what is in that. This is useful as it creates modular code which can be worked on without affecting the overall program due to it calling on the interface, as source code does not need to be changed. It also allows each programmer a template that must be followed within the classes, allowing sub-division of the code to be worked on by multiple programmers without needing to know what the other is doing. For this reason, the code within the program is standardized and enforced through the interface as without the right links the program will throw errors at compilation.



2.3.1 Abstraction

Abstraction is a concept that allows the separation and hiding of the data. It was referred to in section 2.2 as data encapsulation allows abstraction through hiding data. Interfaces can take this concept a stage further. It is possible to take some of the properties and methods from a class and allow public access modifiers within the interface, but it is also possible to just take the method and property signatures needed, which hides the unnecessary information of the class from the user or caller. As an example, using the Car class from earlier, the Driver object would need all the methods within the class and so an interface showing the signature for all of these could be provided to the Driver. However, a BrakePedal object only needs to know if the car accelerates and when to brake so only the Accelerate() and Brake() signatures need to be shown in this interface as the SoundHorn() method is irrelevant and therefore is hidden away from the BrakePedal object or class. In this way it allows a useful coupling with only the relevant information available to the implementor or caller thus subdividing the class for use in different ways. [7] [8]

Consider the code below using the same car class as earlier but adding two new classes and the interfaces:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Example
6 {
7     public interface ICar
8     {
9         void SoundHorn();
10
11         void Accelerate();
12
13         void Brake();
14     }
15 }
16
17
```

Figure 8 - ICar Interface used by Driver

```
1 using System.Collections.Generic;
2 using System.Text;
3
4 namespace Example
5 {
6     public interface ICarBrake
7     {
8         void Accelerate();
9
10         void Brake();
11     }
12 }
13
14
15
```

Figure 9 - ICarBrake interface used by BrakePedal

```

1  using System;
2      using System.Collections.Generic;
3      using System.Text;
4
5  namespace Example
6  {
7      3 references
8      class Car : Icar, ICarBrake
9      {
10         //Attributes
11         private string _model;
12         private string _numberOfDoors;
13         private string _colour;
14         private string _engineSize;
15     }

```

Figure 10 - The Car class showing how to link the interfaces

3.0 Conclusion

OOP is an important, popular and useful paradigm within the higher-level application programming as it allows flexible and modular programming, allowing subdivision of code through classes and interfaces, meaning code can be written separately and brought together to complete a program. The concepts within the paradigm are what make it so versatile with the use of classes to create data types that form objects that can be simulated from real life, interfaces that will allow a link and contract between classes, and through these, the concept of data encapsulation and abstraction, giving the ability for classes to operate without the knowledge of what is in other classes.

This report touches on the fundamentals of OOP. Although there is so much more to learn within this topic, understanding the fundamental concepts, as discussed above, allows an ability to program using DRY (don't repeat yourself) coding, simulate real world items in objects, and the ability to change code without affecting the whole program.

Future research would read more into polymorphism and creating sub-classes from super-classes and the role of interfaces, encapsulation and abstraction within this concept.

References

- [1] R. Yilmaz, A. Sezgin, S. Kurnaz and Y. Z. Arsian, "Object-Oriented Programming in Computer Science," in *Encyclopedia of Information Science and Technology*, M. Khosrow-Pour, Ed., Hershey, PA: IGI GLOBAL, 2018, pp. 7470 - 7480.
- [2] "C# - What is OOP?," W3Schools.com, [Online]. Available: https://www.w3schools.com/cs/cs_oop.asp. [Accessed February 2020].
- [3] "What is the difference between classes and objects?," W3Schools, [Online]. Available: <https://www.w3schools.in/java-questions-answers/difference-between-classes-objects/>. [Accessed February 2020].
- [4] G. G. A. Ganesh, "C# Encapsulation," c-sharpcorner.com, 7th June 2019. [Online]. Available: <https://www.c-sharpcorner.com/article/encapsulation-in-C-Sharp/>. [Accessed February 2020].
- [5] "C# Encapsulation with Examples," tutlane, [Online]. Available: <https://www.tutlane.com/tutorial/csharp/csharp-encapsulation>. [Accessed February 2020].
- [6] "Encapsulation in C#," [Online]. Available: <https://dotnettutorials.net/lesson/encapsulation-csharp/>. [Accessed February 2020].
- [7] A. Stellman, J. Greene, J. Grant and J. A. H. Culby, *Head First C#, 3rd ed.*, Beijing: O'Reilly, 2013, pp. 293 - 318.
- [8] W. Harrison and H. Ossher, "Subdivided procedures: a language extension supporting extensible programming," in *Proceedings. 1990 International Conference on Computer Languages*, New Orleans, 1990.

Appendix

FINAL GRADE

75/100

GENERAL COMMENTS

Instructor

Part 1 :

Class (10): 10

Encapsulation (10): 10

Interface (10): 10

Part 2:

Class (10): 10

Logic (10): 10

Polymorphism (30): 25

Part 3:

Written (10):

Code (10):

Comments

=====

Excellent discussion of concepts (please add a few more references)

Part 2:

Excellent code demonstrating good separation of concerns and use of OOP.

You should always provide evidence of code testing..hence the few marks you dropped.

Also provide a few more comments, they were a bit sparse..



Comment 1

reference

PAGE 4

PAGE 5

PAGE 6

PAGE 7



Comment 2

reference

Good analogy

PAGE 8



Comment 3

Very good!!

PAGE 9

PAGE 10

PAGE 11

PAGE 12
