IT4100 – Software Quality Assurance

Assignment 2 – Cyclomatic Complexity Metric

Group ID – 2022-SQA-G32

Group member details:

Register Number	Name
IT18540536	Lanerolle T.Y
IT18218572	Prashadika W.M.J
IT18222982	Kumarasiri T.S.N
IT18062120	Wickramasinghe W.A.P.C

Declaration

We certify that this report does not incorporate without acknowledgement, any material previously submitted for a degree or diploma in any university, and to the best of our knowledge and belief it does not contain any material previously published or written by another person, except where due reference is made in text.

Table of Contents

Table of Tables

Table of Figures

1. Introduction to Cyclomatic Complexity (CC) metric

Cyclomatic Complexity in Software Testing is a testing metric used for measuring the complexity of a software program. It is a quantitative measure of independent paths in the source code of a software program. Cyclomatic complexity can be calculated by using control flow graphs or with respect to functions, modules, methods, or classes within a software program. Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths. This metric was developed by Thomas J. McCabe in 1976 and it is based on a control flow representation of the program. Control flow depicts a program as a graph which consists of Nodes and Edges.

2. Proposed enhancement factors

2.1. Based on Coupling Between Object Classes (IT18540536 – Lanerolle T.Y)

McCabe cyclomatic complexity is not considering or measuring the exact software complexity means if there is interaction between two or three object classes in software then it does not calculate that complexity. There is a need to consider that object coupling complexity in McCabe cyclomatic complexity. Coupling can occur among object classes through different methods like: Field accesses, through methods calls, Inheritance, Arguments, Return types, Exception, instruction type. Here the examples are based on Inheritance.

Proposed new metric

Given here is a description of how the factors considered in the new metric affects to the complexity of a program.

Coupling between object classes

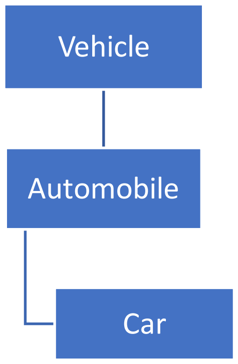


Figure 1.0 Multilevel Inheritance

Multilevel Inheritance	
Base Class	Vehicle
Intermediate Class	Automobile
Derived Class	Car

Table 1.0 Multilevel Inheritance Classes

Coupling between classes has a higher impact on the complexity of a software. As the coupling level goes higher in a code segment it provides too complexity to the program. Effect of coupling level of a class is taken by assigning weights to each level. Starting from the base class by giving weight 1. Likewise, for the child classes, values are given in increasing order.

According to the coupling factor discussed above we came up with a modified Cyclomatic Complexity equation.

New Cyclomatic complexity = Cyclomatic Complexity + Coupling between object classes

Given below are some examples to explain how the complexity is calculated from the metrics.

Suppose the sample program is,

```
Console.WriteLine("Program start");                     a
bool isHavingWheels = true;                              b
if (isHavingWheels)                                      c
{
      Console.WriteLine("Having wheels");                d
}
else
{
      Console.WriteLine("Do not have wheels");           e
}
Console.WriteLine("End of program");                     f
```

Control Flow Graph

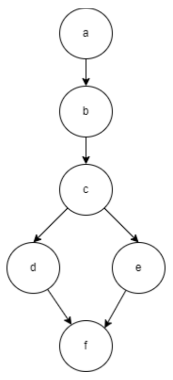


Figure 1.1 Control Flow Graph

v(g)= (d + 1)

where,
d = the number of decision statements

v(g new)= (d + 1) + C

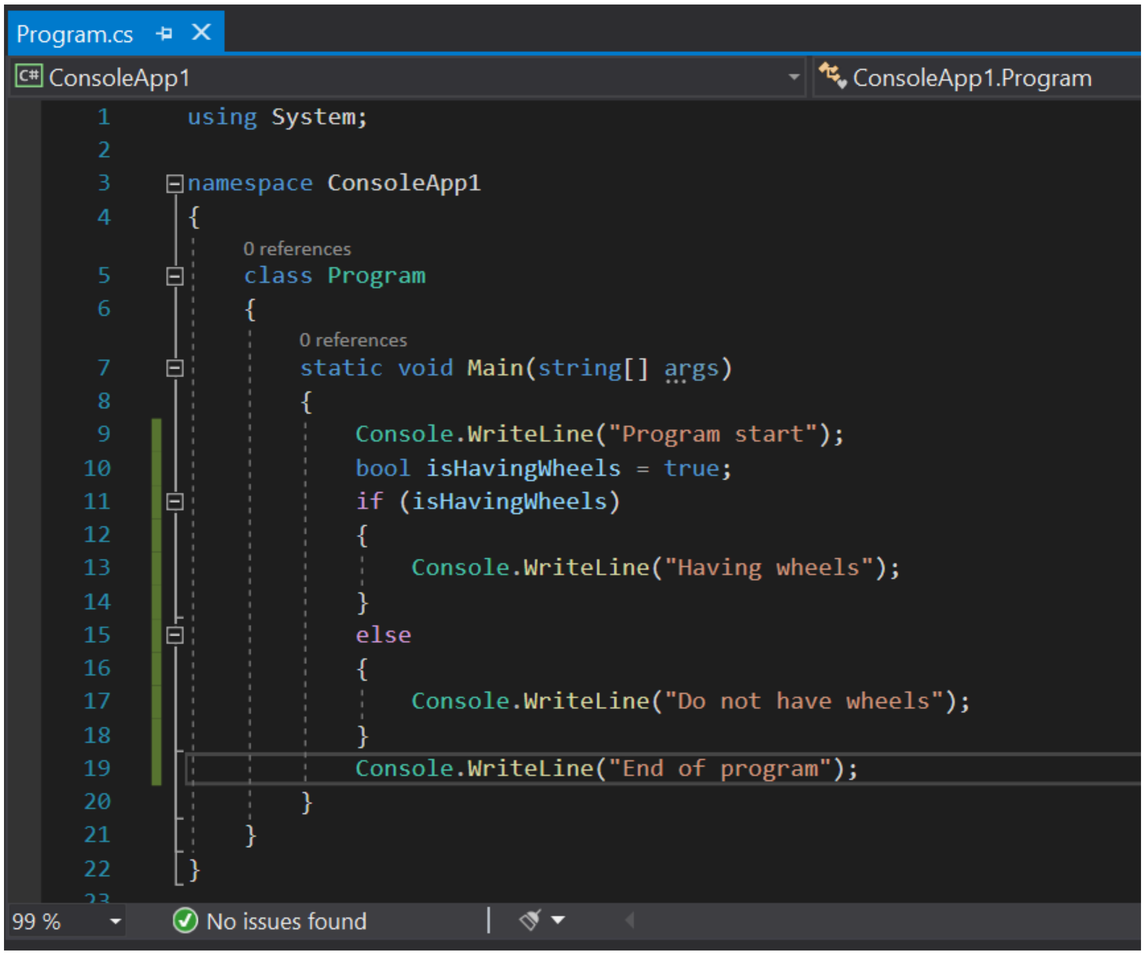
where,
d = the number of decision statements

C = coupling between object classes

v(g) = 1 + 1 = 2

v(g) new = v(g) + 0 = 2

Example 01



Program.cs

ConsoleApp1 | ConsoleApp1.Program

```
using System;

namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Program start");
            bool isHavingWheels = true;
            if (isHavingWheels)
            {
                Console.WriteLine("Having wheels");
            }
            else
            {
                Console.WriteLine("Do not have wheels");
            }
            Console.WriteLine("End of program");
        }
    }
}
```

99 % | No issues found

Figure 1.2 Example 01 Program Class

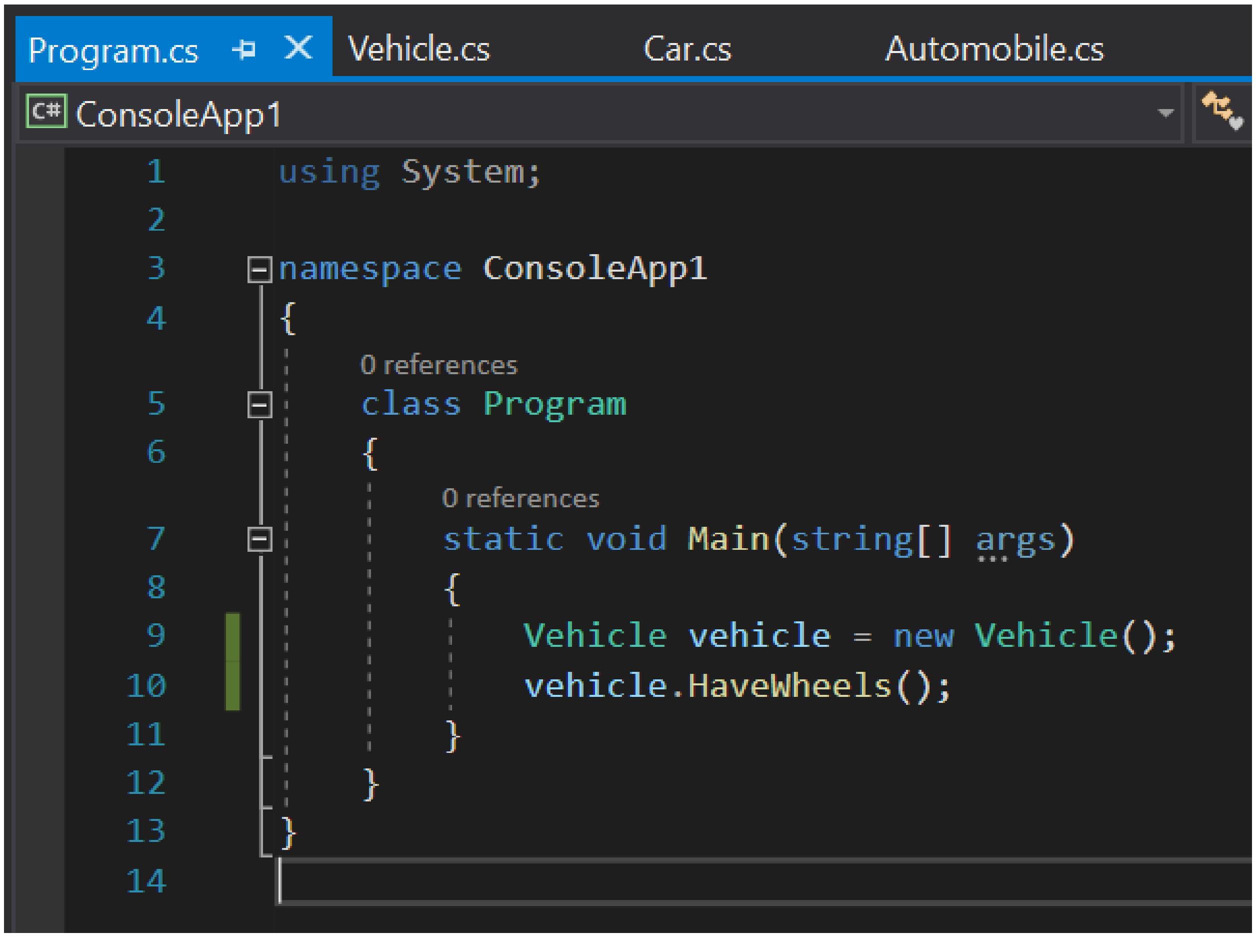
v(g)= (d + 1)

v(g) = 1 + 1 = 2

v(g new)= (d + 1) + C

v(g) new = v(g) + 0 = 2

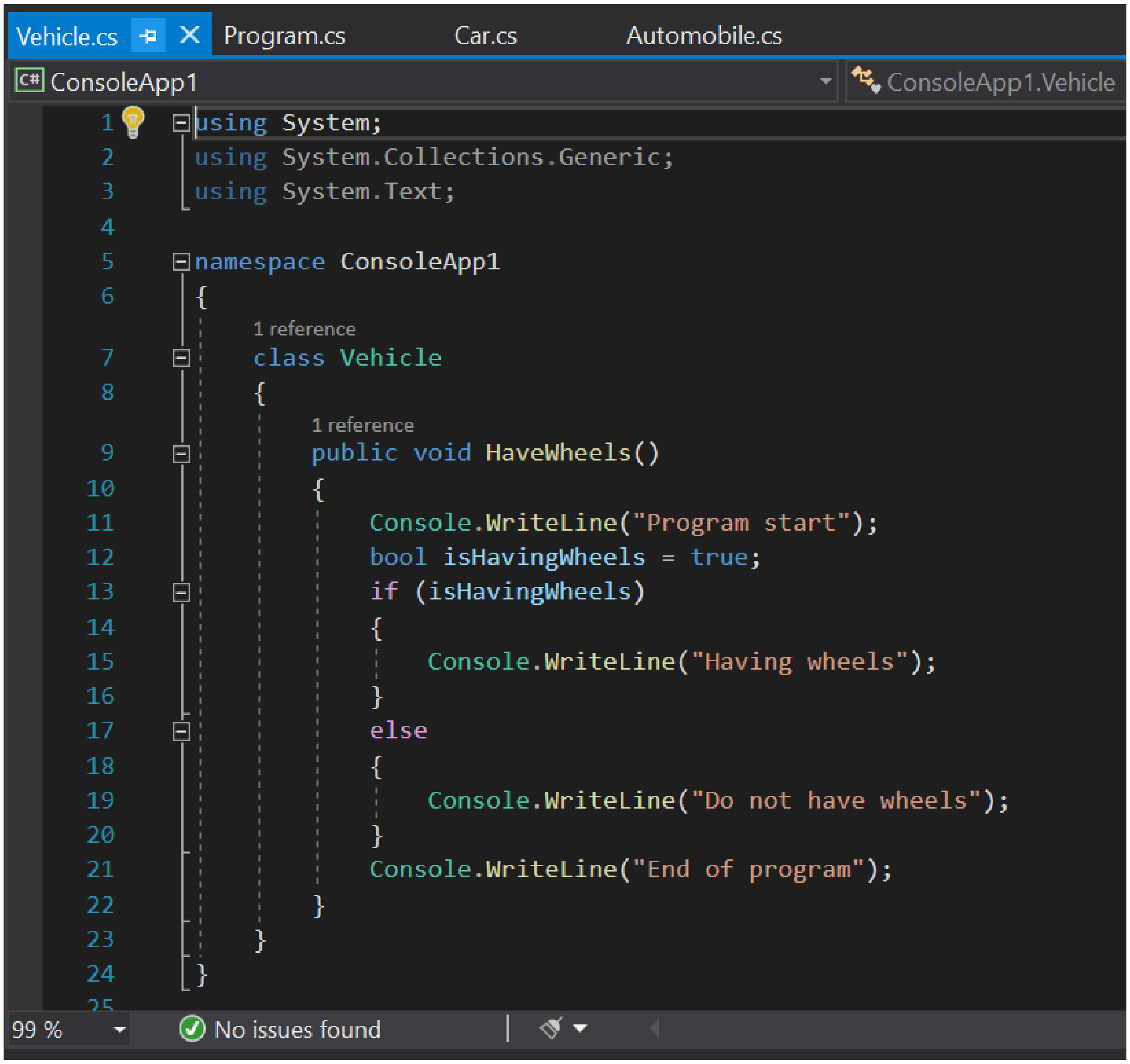
Example 02



```
using System;

namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Vehicle vehicle = new Vehicle();
            vehicle.HaveWheels();
        }
    }
}
```

Figure 1.3 Example 02 Program Class



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

namespace ConsoleApp1
{
    class Vehicle
    {
        public void HaveWheels()
        {
            Console.WriteLine("Program start");
            bool isHavingWheels = true;
            if (isHavingWheels)
            {
                Console.WriteLine("Having wheels");
            }
            else
            {
                Console.WriteLine("Do not have wheels");
            }
            Console.WriteLine("End of program");
        }
    }
}
```

Figure 1.4 Example 02 Vehicle Class

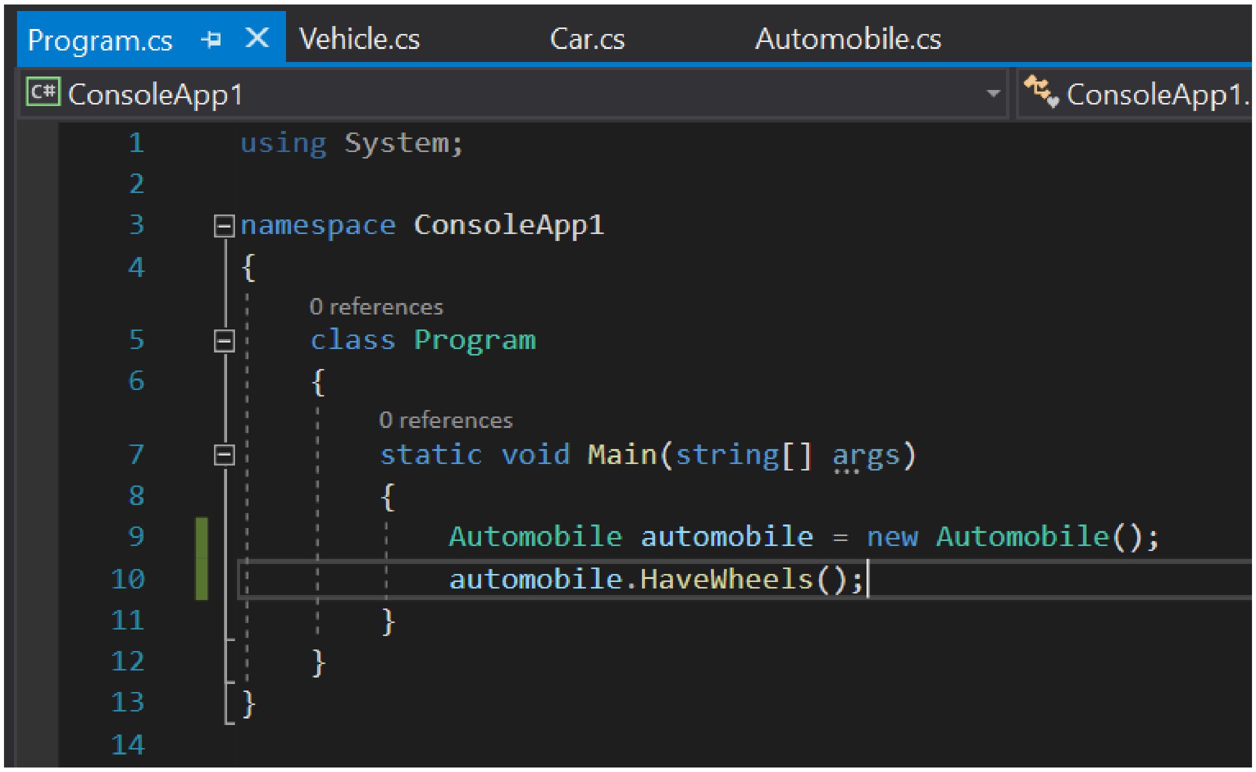
v(g)= (d + 1)

v(g) = 1 + 1 = 2

v(g new)= (d + 1) + C

v(g) new = v(g) + 1 = 3

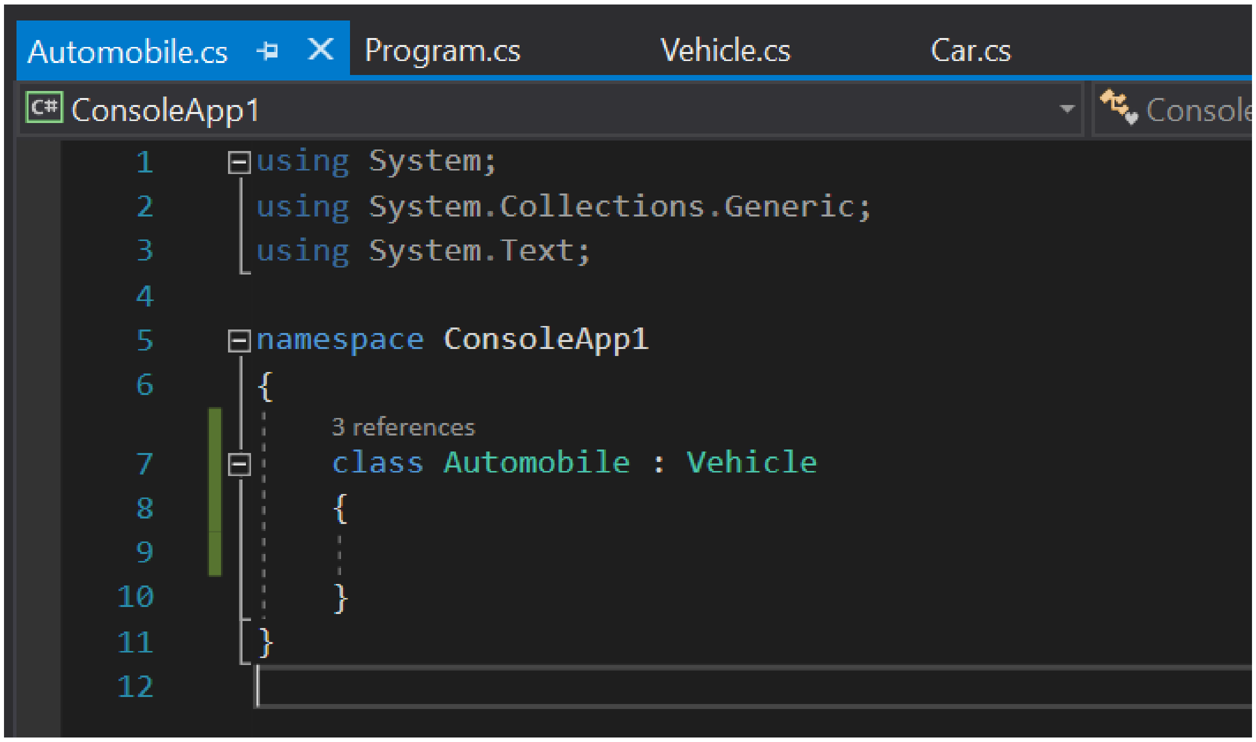
Example 03



```
using System;

namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Automobile automobile = new Automobile();
            automobile.HaveWheels();
        }
    }
}
```

Figure 1.5 Example 03 Program Class

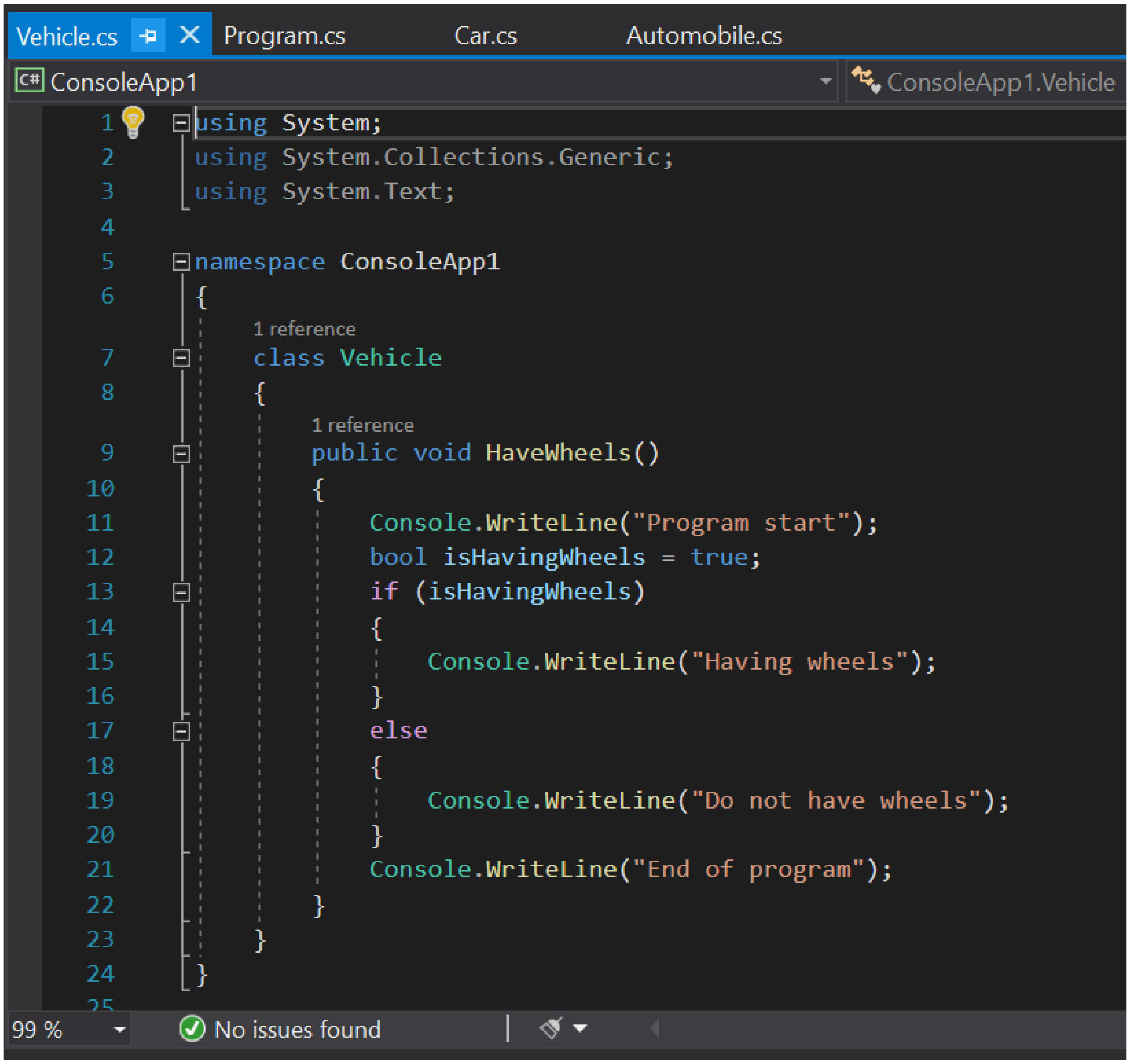


```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApp1
{
    3 references
    class Automobile : Vehicle
    {

    }
}
```

Figure 1.6 Example 03 Automobile Class



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

namespace ConsoleApp1
{
    1 reference
    class Vehicle
    {
        1 reference
        public void HaveWheels()
        {
            Console.WriteLine("Program start");
            bool isHavingWheels = true;
            if (isHavingWheels)
            {
                Console.WriteLine("Having wheels");
            }
            else
            {
                Console.WriteLine("Do not have wheels");
            }
            Console.WriteLine("End of program");
        }
    }
}
```

Figure 1.7 Example 03 Vehicle Class

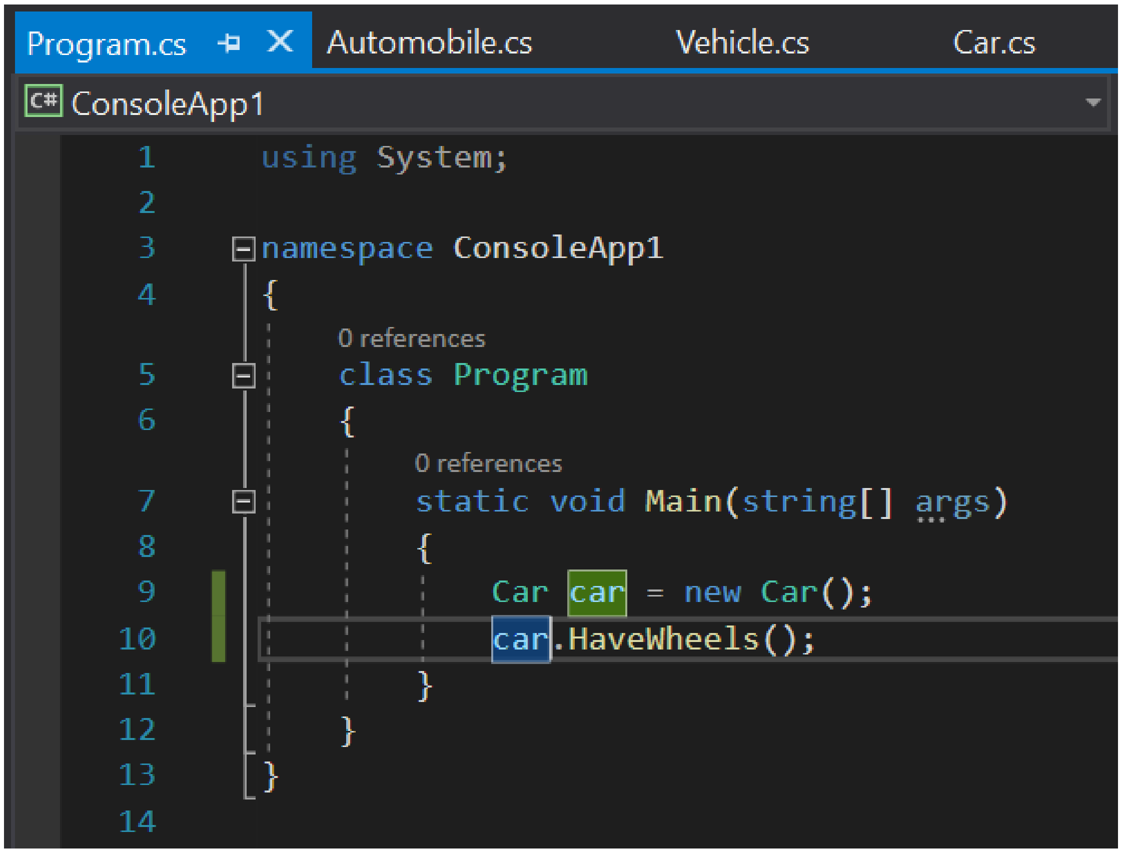
v(g)= (d + 1)

v(g) = 1 + 1 = 2

v(g new)= (d + 1) + C

v(g) new = v(g) + 2 = 4

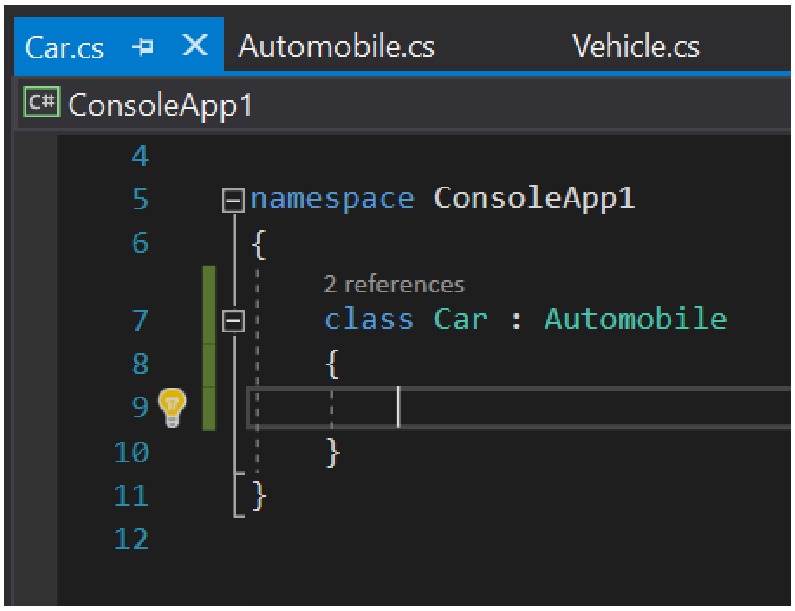
Example 04



```
using System;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Car car = new Car();
            car.HaveWheels();
        }
    }
}
```

Figure 1.8 Example 04 Program Class

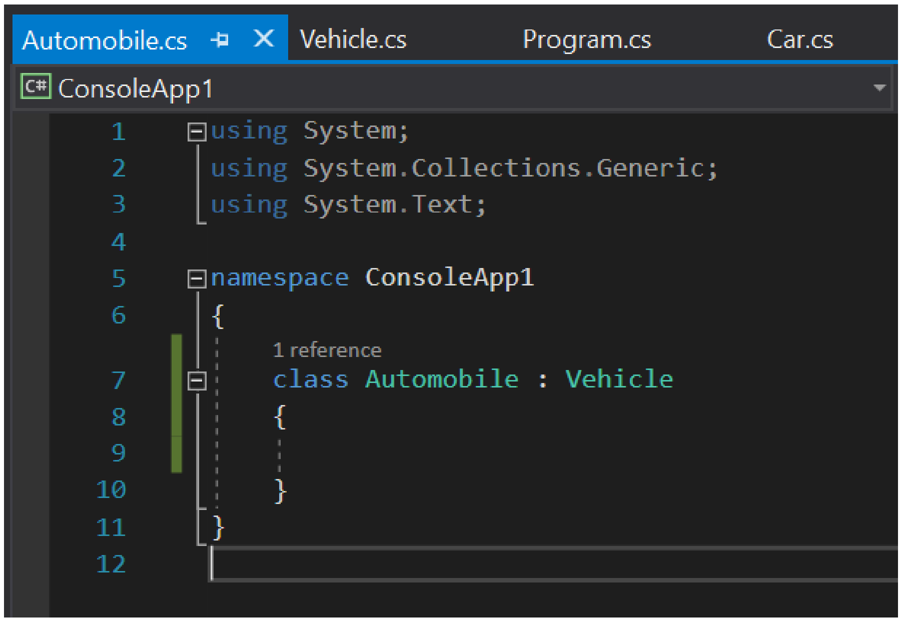


```

namespace ConsoleApp1
{
    class Car : Automobile
    {

    }
}
```

Figure 1.9 Example 04 Car Class


Automobile.cs Vehicle.cs Program.cs Car.cs

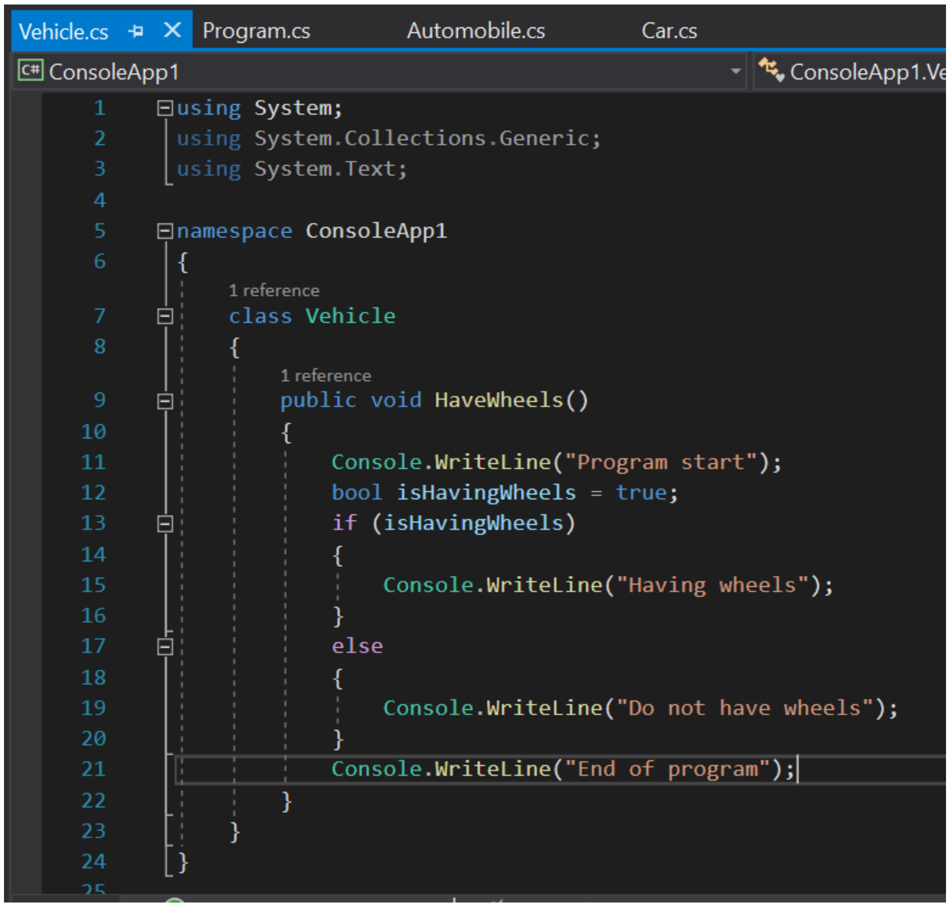
ConsoleApp1

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

namespace ConsoleApp1
{
    1 reference
    class Automobile : Vehicle
    {

    }
}
```

Figure 1.10 Example 04 Automobile Class


Vehicle.cs Program.cs Automobile.cs Car.cs

ConsoleApp1 ConsoleApp1.Ve

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

namespace ConsoleApp1
{
    1 reference
    class Vehicle
    {
        1 reference
        public void HaveWheels()
        {
            Console.WriteLine("Program start");
            bool isHavingWheels = true;
            if (isHavingWheels)
            {
                Console.WriteLine("Having wheels");
            }
            else
            {
                Console.WriteLine("Do not have wheels");
            }
            Console.WriteLine("End of program");
        }
    }
}
```

Figure 1.11 Example 04 Vehicle Class

v(g)= (d + 1)

v(g) = 1 + 1 = 2

v(g new)= (d + 1) + C

v(g) new = v(g) + 3 = 5

2.2. Nesting level **(Prashadika W.M.J-IT18218572)**

Levels of layering are a significant contributor to our complexity metric. As the depth of nesting in a code segment increases, the program becomes increasingly complicated and difficult to understand. Nesting in the control structure is simulated by giving values to the nesting levels. We assign value 1 to the highest level, value 2 to the next highest level, and so on. Values are not provided for sequential statements. Software science and cyclomatic complexity are two well-known measurements of program complexity. Structured vs. unstructured programs, nested vs. sequential predicates, and the usage of case statements are three areas where these measurements may not necessarily fit our intuitive perceptions of complexity. This study proposes a nesting level complexity metric that penalizes unstructured Ness and predicate nesting while rewarding the inclusion of case statements.

2.2.1 Reasons for selection

There are many loops in a Software program such as,

While, DO-WHILE, FOR.

A loop in a computer program is an instruction that repeats until a specified condition is reached. In a loop structure, the loop asks a question. If the answer requires action, it is executed. The same question is asked again and again until no further action is required. Each time the question is asked is called an iteration. Even though these loops reduce lines of codes this may take time to produce an output base on the size of the data source or no of iterations to achieve a task.

2.2.2 Definitions of loops

A for loop is a loop that runs for a preset number of times. A while loop is a loop that is repeated as long as an expression is true. An expression is a statement that has a value. A do while loop or repeat until loop repeats until an expression becomes false. Whenever there is an inner loop within the loop it might makes more complex to occur rather than using simple loops If the programmer didn’t use correct loop controls this programmers' may be run to an endless loop(infinite).

2.2.3 Modification of Cyclomatic Complexity metric

V(g)=(d+1) + (L+(IL/L))

V(g) = (E – N + 2p) + (L+(IL/L))

d=predicted nodes

IL=Total number of Inner loops in loop

L=Total number of Loops with the program

2.3. Number of Inputs and Outputs (IT18222982 - T.S.N kumarasiri)

Inputs and outputs are a very important factor to be considered in a software program where the entire program depends on. Without inputs and outputs, a definite program structure is practically impossible. Number of inputs and outputs of a program can be proposed as a factor to enhance the Cyclomatic complexity. In addition to the identification of the no. of linearly independent paths, CC metric could be modified in a way to identify the inputs and outputs in a program and how they relate with each other.

Selected reason

- Gives a clear definition and structure to the program
- Estimates for a definite outcome
- Provides a higher contribution for the program
- Higher the inputs, higher the allocated memory space and computational power
- Higher execution time with increasing number of inputs and outputs
- High testability with high effort and cost
- Provides clarity and understandable

CALCULATION OF NUMBER OF INPUTS AND OUTPUTS

As inputs consider:

i. Keyboard inputs
ii. Parameters
iii. Data Types

As outputs consider:

i. Return statements (Eg: printf, S.O.P)

CALCULATION OF NUMBER OF INPUTS

- For the data types, we could assign a weight since many data types do exist

Data Types	Weight
Primitive data types: Integer String Double Float Boolean Char	$\left.\begin{matrix}\\ \\ \\ \\ \\ \end{matrix}\right\}$ 1
Complex data types Single dimensional arrays Multi-dimensional arrays Array Lists	 2 3 4

Table 2.3.0 Weight of Data types

N_i = No. of inputs

N_i = (Weight of Data type * Number of parameters) + Keyboard inputs

CALCULATION OF NUMBER OF OUTPUTS

- For the outputs, we could consider the return statements like:

i. printf
ii. System.Out.Println

N_o = No. of outputs
N_o = (Number of Return statements)

Build an equation using number of inputs and outputs

Finally, using the number of inputs and outputs, an equation could be built. We could add both the number of inputs and outputs together.

$$\boxed{N_{IO} = (N_i + N_o)}$$

NIO = Sum of No. of inputs and outputs
Ni = No. of inputs
No = No. of outputs

MODIFIED EQUATION

As number of inputs and outputs increase the complexity of a program, add the N_{IO} value to the existing equation.

$$\boxed{V(g) = (d + 1) + (N_{IO})}$$

N_{IO} = Sum of number of inputs and outputs

EXAMPLE PROGRAM

```
#include <stdio.h>
  int main() {
        int number1, number2, sum;
        printf("Enter two integers: ");
        scanf("%d %d", &number1, &number2);
        // calculating sum
        sum = number1 + number2;
        printf("%d + %d = %d", number1, number2, sum);
        return 0;
 }
```

- **Complexity due to N_{IO}**

  Ni = (1 * 3) + 2 = 5

  (Since the data type of parameters is integer, the weight is 1)

  No = 2

  NIO = (Ni + No)

  = 5 + 2 = 7

- **Complexity due to modified equation**

  V (g) = (d + 1) + (NIO)

  Since there are no predicate nodes, d=0

  = (0 + 1) + 7

  = 8

  Proposed factor, Number of inputs and outputs relate with the complexity metric, where a relationship could be built among each other as shown in the modified equation.

$$V(g) = (d + 1) + (N_{IO})$$

2.4. Prefer Smaller Functions (IT18062120 – W.A.P.C. Wickramasinghe)

All else being equal, smaller functions are easier to read and understand. They're also less likely to contain bugs by virtue of their length.
If you don't have too many lines of code, you don't have lots of opportunities for buggy code. The same reasoning applies for cyclomatic complexity. You're less likely to have complex code if you have less code period. So, the advice here is to prefer smaller functions.
For each function, identify their core responsibility. Extract what's left to their own functions and modules. Doing that also makes it easier to reuse code, which is a point we'll revisit soon.
2.3 Complexity calculation formula
Mathematically, for a structured program, the directed graph inside control flow is the edge joining two basic blocks of the program as control may pass from first to second. So, cyclomatic complexity M would be defined as,

$M = E – N + 2P$

E = the number of edges in the control flow graph

N = the number of nodes in the control flow graph

P = the number of connected components

Steps that should be followed in calculating cyclomatic complexity and test cases design are:

- Construction of graph with nodes and edges from code.
- Identification of independent paths.
- Cyclomatic Complexity Calculation
- Design of Test Cases

section of code as such:

```
A = 10
  IF B > C THEN
    A = B
  ELSE
    A = C
  ENDIF
Print A
Print B
Print C
```

Control Flow Graph of above code

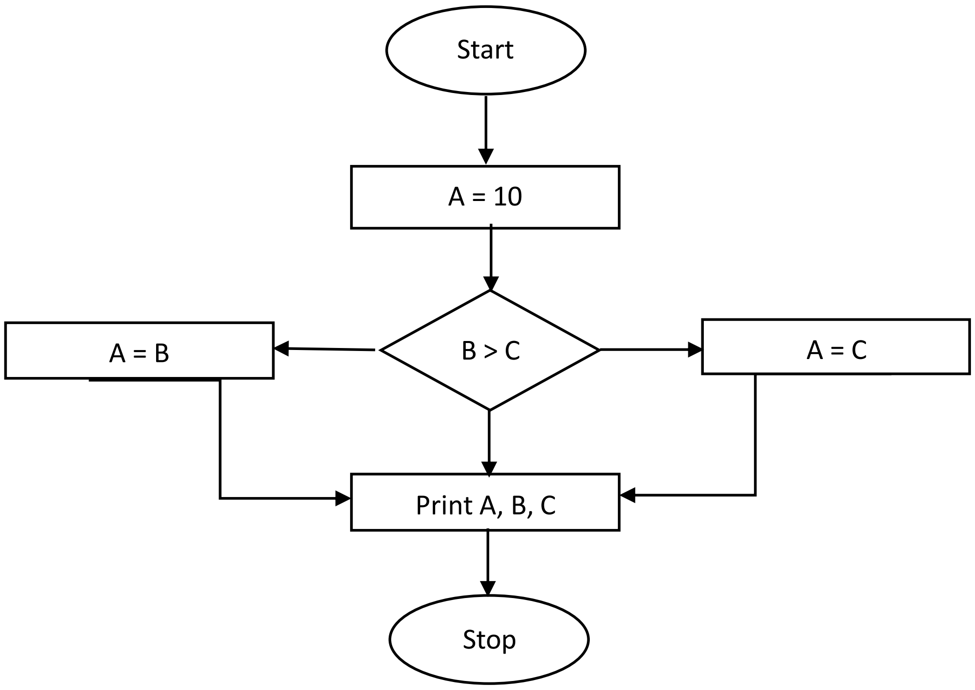


Figure 2.4.1 Control Flow Graph

3. Final modified equation by considering all enhancement factors proposed

Proposed equation contains factors which can be merged to extend the complexity equation.

New Cyclomatic complexity = Cyclomatic Complexity number + Newly Added Factors

$v(g)= (d + 1)$

v(g new)= (d + 1)+F1+F2+F3

F1 - Based on Coupling Between Object Classes

F2 - Nesting level

F3 – Number of Inputs and Outputs

Final Form

$$V(g\ new) = (d+1) + C + (L+(IL/L)) + N_{IO}$$

d = the number of decision statements

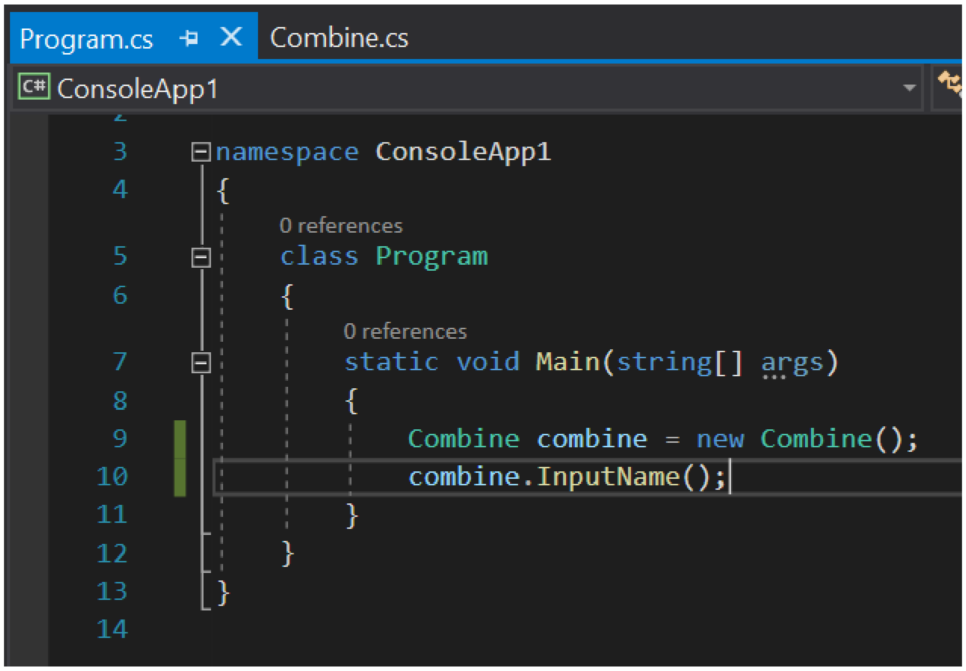
C = coupling between object classes

L = total number of Loops with the program

IL = total number of Inner loops in loop

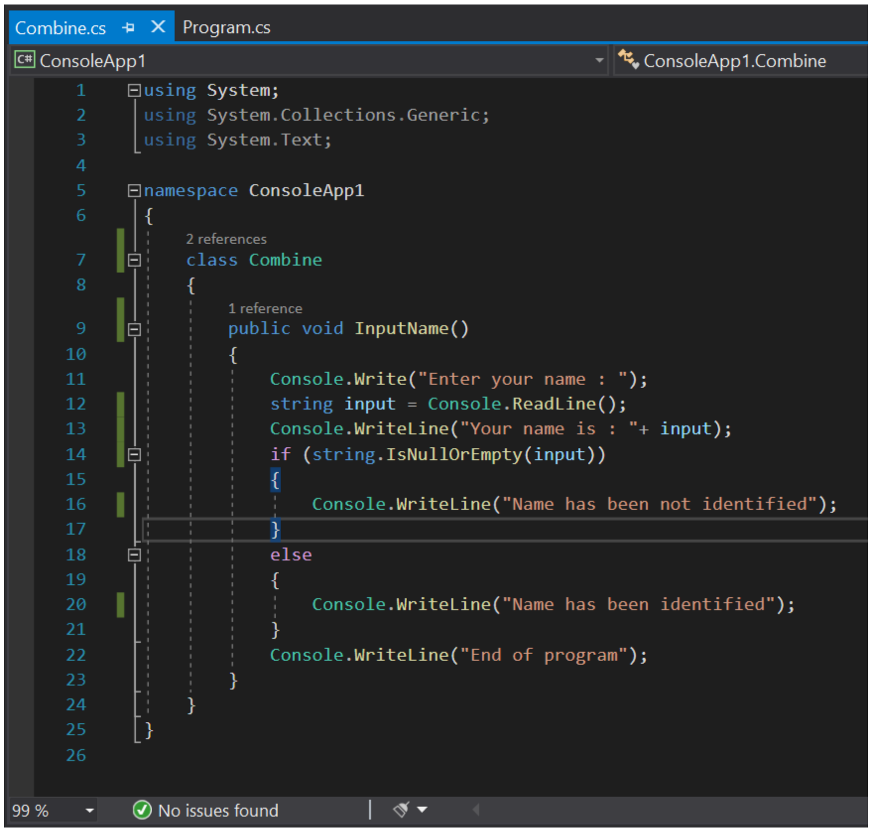
N_{IO} *= total number of inputs and outputs*

4. Example of calculating the complexity using final modified equation



```
namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Combine combine = new Combine();
            combine.InputName();
        }
    }
}
```

Figure 4.1 Example Program Class



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

namespace ConsoleApp1
{
    2 references
    class Combine
    {
        1 reference
        public void InputName()
        {
            Console.Write("Enter your name : ");
            string input = Console.ReadLine();
            Console.WriteLine("Your name is : "+ input);
            if (string.IsNullOrEmpty(input))
            {
                Console.WriteLine("Name has been not identified");
            }
            else
            {
                Console.WriteLine("Name has been identified");
            }
            Console.WriteLine("End of program");
        }
    }
}
```

Figure 4.2 Example Combine Class

v(g) = (d + 1)

v(g) = 1 + 1 = 2

***v*(g new) = (d+1) + C + (L+(IL/L)) + N$_{\mathbf{IO}}$**

v(g new) = (1+1) + 1 + 0 + 6 = 9

Conclusion

McCabe cyclomatic complexity measure of software complexity is one of the strongest metrics among LOC, Halstead's and language independent. Till now it is used to calculate the software complexity only through control flow graph (via conditional statements) but there is a need to include the concept of coupling with cyclomatic complexity concept and for that one algorithm is purposed, how can calculate the coupling between object classes for object-oriented programming and then, used that concept in cyclomatic complexity for the improvement of cyclomatic complexity concept. Loops cost some complexity if there is an inner loop which directly cost the execution flow. We could assume that the total sum of both the inputs and outputs would increase the complexity of a program since an addition operation ($N_i + N_o = N_{IO}$) is carried out here. Therefore, we could conclude that the complexity increases with the increase of no. of inputs and outputs.

References

[01] B. Souley and B.Bata, “A Class Coupling Analyzer for Java Programs”, West African Journal of Industrial and Academic Research, Vol.7, No. 1, pp. 3-13, 2013.

[02] S.Singh Rathore and A.Gupta, “Investigating Object-Oriented Design Metrics to Predict Fault-Proneness of Software Modules”, 6 th International conference on Software Engineering, pp. 1-10, 2012.

[03] N. I. Enescu, D. Mancas, E. I. Manole, and S. Udristoiu,” Increasing Level of Correctness in Correlation with McCabe Complexity”, International Journal of Computers, Vol. 3, pp. 63-74, 2009.

[04] R . Banker , “Software Complexity And Maintainability”. Philip A. Laplante (25 April 2007). What Every Engineer Should Know about Software Engineering. CRC Press. p. 176. ISBN 978-1-4200-0674-2.

[05] Fricker, Sébastien (April 2018). "What exactly is cyclomatic complexity?". froglogic GmbH. Retrieved October 27, 2018. To compute a graph representation of code, we can simply disassemble its assembly code and create a graph following the rules:

[06] Harrison (October 1984). "Applying Mccabe's complexity measure to multiple-exit programs". Software: Practice and Experience. 14 (10): 1004–1007. doi:10.1002/spe.4380141009. S2CID 62422337.

[07] Arthur H. Watson; Thomas J. McCabe (1996). "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric" (PDF). NIST Special Publication 500-235.

[08] ISO 26262-3:2011(en) Road vehicles — Functional safety — Part 3: Concept phase. International Standardization Organization.

[09] Papadimitriou, Fivos (2012). "Artificial Intelligence in modelling the complexity of Mediterranean landscape transformations". Computers and Electronics in Agriculture. 81: 87–96. doi:10.1016/j.compag.2011.11.009.

[10] Papadimitriou, Fivos (2013). "Mathematical modelling of land use and landscape complexity with ultrametric topology". Journal of Land Use Science. 8 (2): 234–254. doi:10.1080/1747423X.2011.637136. S2CID 121927387.

[11] Les Hatton (2008). "The role of empiricism in improving the reliability of future software". version 1.1.

[12] Kan (2003). Metrics and Models in Software Quality Engineering. Addison-Wesley. pp. 316–317. ISBN 978-0-201-72915-3.

[13] Diestel, Reinhard (2000). Graph theory. Graduate texts in mathematics 173 (2 ed.). New York: Springer. ISBN 978-0-387-98976-1.