# Programming Language

The programming language used to implement the project is C#.

Below are some of the reasons why I chose C#:

1. **Modernized language:**    C# is projected to be a simple, modern, general-purpose, and object-oriented programming language. This presents many benefits. For example, in some object-oriented programming languages, get and set operations are implemented as two different methods. In C#, get and set operations can be implemented in a single property.
2. **Object-orientation**: C# goes beyond the level of object-orientation. Even simple data types like int, datetime, short, etc, can be treated as objects; they have methods associated with them. This was particularly useful in this project during the logging of exceptions. I used ***DateTime.UtcNow.ToString("s")*** to easily write in the ISO 8601 format the UTC date and time at which an exception was thrown.
3. **Easy Memory Management**: C# also removes memory management issues from the developer by using the .NET’s garbage collection scheme. Objects no longer referenced are marked for garbage collection, and the .NET Framework can reclaim this memory as needed. In particular, if an object implementing the ***System.IDisposable*** interface is wrapped in ***using*** statement, that object is automatically marked for garbage collection as soon as the using statement is exited. We no longer need to manually close database connections and file stream objects wrapped in using statements. This feature is particularly useful for this project where file input and output operations are involved. Example of usage can be seen in the ***Statistics.GetHosts*** and ***Statistics.Generate*** methods of the ***HostFleetAnalyzer.Reporting*** project.
4. **Easy to use input/output system**: Due to the .NET Framework, input/output file operations are very easy in C#. This was very critical in the implementation of this project.
5. **Type-safety**: The type-safety of C# is type-safe offers many advantages. For example, uninitialized variables cannot be used. In some object oriented programming languages, it is possible to declare a variable and then check its value; whatever was in the memory address given to that variable would then be shown, and this could have unpredictable results in an application. The C# complier will complain if an attempt is made to use a variable before it is initialized to some valid value.
6. **Cross-language capabilities**: C# has the ability to allow us to interoperate with any other languages on the .NET platform. It is easy to create a component in one language and inherit and extend that component in another language, which is something that was difficult with COM.
7. **Easy Exception Handling**: C# also supports the concept of error handling across different languages. The .NET exceptions are easy to use and are consistent across any .NET language. In particular, in this project, we create a base exception class, namely, ***HostFleetAnalyzer.Reporting.StatisticsException*** from which derives other custom exceptions. With this capability, we can display user-friendly messages to the user when a custom exception we defined is thrown.

# Technology used

The project was implemented with Visual Studio 2013 and the .NET Framework 4.5.

# Assumptions

Below are the assumptions that were made while implementing this project:

1. The FleetState.txt and Statistics.txt files are in the directory of the executable file of the program.
2. The most filled host has at least one occupied slot and one empty slot.
3. The instance type that can run on a host is not case sensitive. Therefore m1 and M1 represent the same instance type.

# Program Structure

The solution consists of three projects, namely, ***HostFleetAnalyzer.Reporting***, ***HostFleetAnalyzer.Reporting.Test***, and ***HostFleetAnalyzer.Terminal***.

1. **HostFleetAnalyzer.Reporting**: This project is a class library. It defines most of the classes used in the application, and is responsible for file input and output operations.
2. **HostFleetAnalyzer.Reporting.Test**: This project is a unit-test project. It automates the testing of the **HostFleetAnalyzer.Reporting** project. It has unit-test methods that are used to test use-cases.
3. **HostFleetAnalyzer.Terminal**: This project is the user-interface of the application.

The project was built with the Release configuration. Therefore, the FleetState.txt and Statistics.txt files are found in the .\bin\Release folder.

# Exception Handling

## Custom Exceptions

To easily handle errors that are likely to occur, we defined the below exceptions in the HostFleetAnalyzer.Reporting namespace of the HostFleetAnalyzer.Reporting project:

**StatisticsException**: This is the base class for all custom exceptions defined in the project.

**HostInstanceTypeRequiredException**: This exception derives from **StatisticsException**. It is thrown when the type of instance that can be ran on a host is not specified on a line in the input file.

**TotalSlotsCountException**: This exception derives from **StatisticsException**. It is thrown when the total number of slots specified for a host in an input file is less than or equal to 0.

**SlotStateFormatException**: This exception derives from **StatisticsException**. It is thrown when the state of a slot in an input file is neither 0 nor 1.

**SlotsStatesMismatchException**: This exception derives from **StatisticsException**. It is thrown when the count of the states specified for a host is not equal to the total number of slots specified for that host. For example, if in the input file, it is indicated that the total number of slots on a host is 5, and the states 0,1,1 or 0,1,1,1,1,1,1 are specified, this exception will be thrown.

**TotalSlotsFormatException**: This exception derives from **StatisticsException**. It is thrown when the total number of slots specified for a host in an input file is not an integer.

**HostIdFormatException**: This exception derives from **StatisticsException**. It is thrown when the Id specified for a host in an input file is not an integer.

**HostLineFormatException**: This exception derives from **StatisticsException**. It is thrown when the line representing a host in an input file is not well formatted.

Any of the above exceptions has a user-friendly message that is displayed on the terminal when the exception is thrown.

## Built-in Exceptions

The program also displays user-friendly messages when the **System.UnauthorizedAccessException** and **System.FileNotFoundException** exceptions are thrown.

The **System.UnauthorizedAccessException** exception is thrown by the .NET Common Language Runtime when the operating system denies access to the location of the input or output file due to lack of enough permissions to that location. If the program (including its input and output files) is ran from a location where the user has enough permissions, this exception will not be thrown. Otherwise, the user will have to be granted enough permissions or the program will have to be ran under-elevated privileges. For security reasons, it is not recommended to run a program under elevated privileges if it can be ran with lower privileges; Viruses, malwares, etc, may use that security breach.

The **System.UnauthorizedAccessException** exception is thrown when an attempt to access a file that does not exist on disk fails. This will happen if the FleetState.txt file is not in the same directory as the program’s executable.

## Exceptions Logging

All thrown exceptions are logged in the %LOCALAPPDATA%\HostFleetAnalyzer.Terminal\log.txt file. The choice of the %LOCALAPPDATA%\ path is because the user will almost always have access to it since it is in his/her profile.

In the log file, the timestamp at which an exception occurred is in UTC and formatted as ISO 8601. For example, an entry in the log file may begin as follows:

*[2015-10-17T09:59:46] System.IO.FileNotFoundException:*

When any other exception not mentioned in this document is thrown, the message ***“An unexpected error occurred.”*** is displayed and the exception is logged.

# Unit-Tests

All unit-test methods are located in the **HostFleetAnalyzer.Reporting.Test.StatisticsUnitTest** class of the **HostFleetAnalyzer.Reporting.Test** project.

## Test Methods Convention

The convention used to name the unit-test methods is as follows:

<*MethodName>\_<InputCondition>\_<Output>*

For example, the test method signature *Generate\_InputFileIsValid\_GenerateValidOutputFile()* indicates the following:

1. The test case under consideration involves the **Statistics.Generate** method.
2. The input file is valid. That is, it has no errors and all the constraints on the host lines have been respected.
3. A valid output file is generated. This is the Statistics.txt file

The test method signature *Generate\_FirstLineInInputFileHasNoInstanceType\_ThrowHostInstanceTypeRequiredException()* indicates the following:

1. The test case under consideration involves the **Statistics.Generate** method.
2. The host Instance Type has been omitted in the first line of the input file (FleetState.txt)
3. The **HostInstanceTypeRequiredException** is thrown*.*

## Test Method input files

The below files have been created to serve as input files for the unit test methods, and can be found in the .\bin\Debug folder of the **HostFleetAnalyzer.Reporting.Test** project:

* ValidInputFile.txt
* InvalidHostIdInputFile.txt
* InvalidHostLineInputFile.txt
* InvalidSlotStateInputFile.txt
* InvalidTotalSlotInputFile.txt
* NegativeTotalSlotInputFile.txt
* NoInstanceTypeInputFile.txt
* NonBinarySlotStateInputFile.txt
* SlotsStatesMismatchInputFile.txt

# Limitations

1. For now, the application has a command-line user interface. The user does not have the choice of the location of the input and output files, and the input file must be named FleetState.txt. This project can be improved with a graphical user interface in which the user can make use of the OpenFileDialog control to choose as input, a text file with any name. This will be more error tolerant. For example, the user may unintentionally not name the input file as expected.
2. The application may also be optimized by localizing it. This will allow users from different cultures to use the program easily.
3. Only three instance types (M1, M2, and M3) were used in the program. We may extend the program to use and number and types of instance types.