# Introduction

This document is intended to support the code produced in respect of the Focus Interview Programming Challenge. It aims to explain key elements of principal importance in the implementation of the requirements and will explain why the code has been designed in the way that it has been implemented. It also acts as a commentary to supplement the code, comments and enables the viewer to more fully understand the thought processes involved in its construction.

|  |  |  |
| --- | --- | --- |
| **Document Title** | **Version** | **Version** |
| Lottery Numbers Generator Design Document | Dave Pugh | 1.0 |

# Document Inputs

The following document(s) have been used as the input for requirements to this design document.

|  |  |  |
| --- | --- | --- |
| **Document Title** | **Author** | **Version** |
| Focus Interview Programming Challeneg | Paul Foster | 1.1 |

# Solution Design

The following table describes the projects utilised within the LotteryNumbersGenerator solution and describes the high level purpose of each project.

|  |  |
| --- | --- |
| **Project Name** | **Purpose** |
| LottoNumberGenerator.UI | The User Interface element of the solution. Calls business logic and displays results with the most minimal amount of logic employed. |
| LottoNumberGenerator.BusinessLogic | Contains the business logic for the application, so that other components such as an ASP.NET MVC or WPF application could use the same logic with very minimal work. |
| LottoNumberGenerator.UI.UnitTests | Contains unit testing (Xunit) of the minimal logic within the UI. |
| LottoNumberGenerator.BusinessLogic.Tests | Contains unit tests (XUnit) to test the logic employed within the business logic layer. |

# Design Philosophy

## Separation of concerns

The solution has been segmented to ensure separation of concerns – namely a UI layer, which only has responsibility essentially for calling the Business Logic tier and displaying the required data and a Business Logic layer. The business logic layer is responsible for implementing the main logic of the application. By doing so it makes the code more testable and also means other interfaces could, in future, make use of the same functionality.

## Inversion of Control/Dependency Injection

The Business Logic layer implements an interface, which allows for the loose coupling of components to it and ensures that dependency injection can be utilised to ensure the inversion of control from within the UI layer. Inversion of control has been implemented within the UI layer, which would allow for a different implementation of the ILotteryNumberGenerator to be used with only 1 slight change to the program.cs file within the UI project.

## Configuration

Configuration has been provided within the UI layer, which allows for the number of required lottery numbers to be generated to be modified without the expense of extra development work, quality assurance and a new production deployment, which also limits the risk associated with such a change.

## Default and overridden implementation

Two overridden methods exist within the business logic class to generate the lottery numbers, one to perform the default requirement of producing 6 lottery numbers, the other accepts a parameter whereby the number of lottery numbers required can be passed in, such as 7, which is a potential future requirement. Therefore this behaviour again allows for future change without changes to the business logic of the application, again reducing risk of such a change in functionality.

## Designing the business logic around the TextColour enum

A more simplistic approach to this requirement may have been to just implement static code to implement the required colouring of text, however this would mean that in future such logic would need to change each time a new colour was added for instance. Therefore the logic has been intrinsically linked to the TextColour enum, which also holds values for the upper value of each text colour, for instance Grey = 9. By doing so all that would be needed should a new colour need to be introduced, is the modification of the TextColour enum – no further business logic code change would be required – again reducing risk of such a change. This approach has been tried and tested with both manual and unit tests. Also by linking the logic to the enum, any removal or improper modification of that enum would see immediate compile time errors, rather than potential issues which may only get caught within the quality assurance effort or worse still production, which always prove more expensive to fix.

## Business Logic not returning ConsoleColor enum

It would have been easy for the business logic to return ConsoleColor as part of its returned object rather than the custom TextColour enum. The reason being that, if ConsoleColour would have been returned there would be no need for a converter in the UI layer to convert TextColour to ConsoleColor. However such an approach would have locked the business logic into a Console Application implementation and therefore limited its further extended use. By instead using the TextColour enum, in future other front ends can call this business logic and implement a simple converter to display the colours as per their requirements. This makes the software more open to future use and in a real world scenario would improve Return On Investment (R.O.I).

## Unit Testing

Unit testing has been included to ensure the correct functioning of the product. Testing of private methods has been implemented, which is not always advised, however in this instance I believe the testing of such methods adds a great deal of value, value which is greater than the perceptions of testing non-public code. It also means that the design of the code stays true to what is required and items that should be private aren’t “upgraded” to internal just so that they can be tested directly in unit tests. An example of one of the benefits of testing private methods in this solution is that it enables the private method DetermineTextColour(int number) to be called and tested directly and therefore for each boundary i.e. 1-9, 10-19 to be rigorously checked – this would not be able to happen otherwise as it would be called from the ﻿public GenerateLotteryNumbers method with a set of random numbers, which would differ each time – a worthy trade-off I believe.

Tests utilise the Arrange, Act, Assert notation to make it easier for future developers to easily break the tests apart into their constituent required steps. Tests also take the form of UnitOfCodeUnderTest\_TestScenario\_ExpectedResult

## Quality Assurance

Quality assurance has been carried out to ensure that the product meets the needs of the product owner, both with manual testing and also the creation of unit tests, which will also prove beneficial as a sense check and regression test if future changes are required for this product.

## Manual Test Evidence

The following tests all ensure the following as a base set of required outcomes:

* Results are numerically ordered
* Results have the correct colour coding
* Numbers returned are unique within that set

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Evidence** | **Outcome** |
| Default functionality of app – ensure 6 numbers are returned | ﻿ ﻿ | Pass |
| Future functionality (7) – ensure 7 numbers are returned |  | Pass |
| Edge case test for future functionality (10) – ensure 10 numbers are returned |  | Pass |

## Unit Test Evidence

The following is the evidence of unit testing completed against the LottoNumberGenerator.BusinessLogic and LottoNumberGenerator.UI projects. As can be seen from the evidence all tests are passing. Further information as to the objective of each test can be found within the comments of each test within the test projects.

