### Design and Approach Document for Technical Project Test

#### Eugene Ordeniza

Within this document, I will outline the approaches I took to get to the current project. Along with the show of works and alternative solutions, I will discuss why they were not chosen and which options I ultimately decided to work with.

#### Preamble:

C# ( .NET) was chosen for this project, as it was my most familiar language. I had not used Blazor before this project test; however, I was eager to utilise Blazor’s ability to create “code-behinds”. Allowing me to create web applications that join HTML and CSS, with the rigidity of C#, within the same reference.

Ultimately, I had chosen to work with C# and the Blazor Framework to test my ability to learn and absorb new information – but also to test my coding skills in C# and unfamiliar frameworks or technologies.

#### Background:

Given the nature of the problem, I initially considered how each digit, or what I referred to as “segments,” would be handled. As transferable digits and their wording correspond to an ordinal “set” that can be done in any language.

For example, given a number:

A pseudocode rendition would translate as such:

A pointer can be called for the numerical, tenth, and hundredth place.

Would result in:

As such, consideration was made to discern how the solution would separate the segments and how efficient, readable, and compact the solutions were. In addition, whilst an AI API can be integrated to easily translate a number through a prompt, because of the nature of this technical test, it was not considered.

#### First Implementation – Brute Force Approach

The first iteration of the conversion function intended to apply the theory outlined above; however, it intended to split the number into three workable digits using the math operators outlined below:

This approach would allow an iterative approach over the whole number, by taking the last three 3 numbers and removing them from the original instance. As such, while the number is not empty or zero, it can translate 3 digits at a time. Appending a “thousand” or “million”, depending on how many iterations were translated; 2 iterations, for example, would mean the number is within the thousands, etc.

##### Efficiency

This solution was not chosen for the submission, as its approach to the number conversion was not very efficient. Unit tests are built and compiled in an average of ~3 seconds, over 59 tests, but often struggle on longer numbers, such as trillions and upwards. Furthermore, due to the nature of the iterated number, the preparation of the number was imperative – it needed to not be negative before conversion could occur.

##### Code Readability and Maintainability

I was further not enthused by this approach, as its readability was overall worse, and I knew I could improve on its readability and maintainability. This solution initially was over 110 lines, with heavy use of “if’s”, clumsy as it needed to trim output strings, and needed to declare attributes on the number to append in the end step.

##### Second Implementation – Array-based approach

The second implementation of the conversion function utilises arrays to separate the numbers into workable segments, then iterate over the array, utilising a similar translation algorithm but allowing for a compact and readable approach.

Translating over an array, the configuration would only care about translating objects under a hundred. Then, depending on the number of segments in the array, it would append the respective marker for values over a thousand, a million, or more.

The benefit of this approach would be its readability; clear declaratives are easier to follow, with the word being stored and built in a list string. Then, output all at once without the recursion implementation in the function itself.

##### Efficiency

This implementation ran 59-unit tests on average, ~1.6 seconds. Whilst struggling with larger numbers at a trillion or higher, the array approach did not need to iterate over the whole number in comparison to the brute-force approach. Instead, iterate over the array once, then reverse the translated lists, then combine them for output.

##### Code Readability and Maintainability

This implementation was much more readable than the previous approach; however, readability could still be improved. The conversion that iterated over the array was largely clumsy and was much harder to read and maintain.

##### Current Approach and Ditching Segmentation Approach

This approach utilises the string-based approach that was visited within the brute-force implementation. What both previous approaches lacked in common was compactness and a strong OOP approach that would allow improved readability and maintainability.

Whilst an array and list approach was easier to follow for another developer, it utilised too many declarative variables along with list manipulation. I compacted “If” statements into their smallest form and avoided iterating over non-imperative arrays or lists. Making the function recursive allowed for improved succinctness. Achieving a much more readable and maintainable function, whilst minimising the majority of complexities attempted in previous approaches involving segment handling or separation.

Dollars and Cents were not handled within the function itself, because the conversion function may be recalled for non-currency-related uses. As such, dollar and cent notation are added to the display or output within the webpage.