Design and Approach Document for Technical Project Test

Eugene Ordeniza

This document outlines the design rationale, development process, and technical approaches used in building the Number to Words Web Page project. It also discusses alternative methods considered, the reasoning behind the final implementation, and opportunities for future improvement.

Preamble:

For this project, I chose C# (.NET) as my main language because I am familiar with it and it offers strong support for structured, maintainable code. Although I had not used Blazor before, I saw this exercise as a chance to expand my skills and explore its component-based architecture and "code-behind" model — enabling seamless integration of C#, HTML, and CSS within a single project reference.

My objective was to demonstrate adaptability in learning a new framework while showcasing sound software engineering practices in C#. The project, therefore, became both a technical solution and a self-assessment of my ability to apply familiar programming principles in an unfamiliar environment.

Background:

The challenge was to translate a numerical input into its corresponding English word representation. Conceptually, each number can be decomposed into **segments** (hundreds, thousands, millions, etc.), each mapped to a textual equivalent.

For example, given a number:

$$number = 123$$

A pseudocode rendition would translate as such:

$$Given: Tens = \{Twenty, Thirty, \dots\}, Digits = \{One, Two, \dots\}$$

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

$$Translation = Digits\{1\} + \text{"Hundred"} + Tens\{1\} + \text{"-"} + Digits\{3\}$$

Would result in:

$$Translation = One\ Hundred\ and\ Twenty - Three$$

This segmental structure underpinned all subsequent approaches, but not the final approach. While external APIs or libraries could perform such translations easily, they were excluded per the test requirements to ensure a fully custom implementation.

1. *First Implementation – Brute Force Approach*

The initial version applied a **modulo-based segmentation algorithm** to extract three-digit groups from right to left using mathematical operations:

$$Last\ Three\ Digits = number\ \%\ 1000$$

$$numerical = \frac{numerical}{1000}$$

Each segment was translated individually, and suffixes such as *"Thousand"* or *"Million"* were appended based on iteration count.

Efficiency

While functionally correct, this brute-force method was inefficient for large numbers. Running ~59 unit tests averaged around 3 seconds, with noticeable performance degradation for values above the trillions. Additionally, handling negative inputs required separate pre-processing, increasing complexity.

Code Readability and Maintainability

The implementation suffered from low readability due to extensive conditional branching (if statements) and redundant trimming operations. The function exceeded 110 lines and lacked clear separation of concerns, making long-term maintenance challenging.

This version was ultimately rejected in favour of a cleaner, more modular approach.

2. Second Implementation – Array-based approach

The second design introduced arrays to segment numbers more efficiently. By grouping digits into arrays, each segment (e.g., hundreds, thousands) could be processed independently and then recombined for final output.

$$Example: number = 12345, to\ number\ segments = [345, 12]$$

Each segment was translated using consistent logic, and the corresponding scale (Thousand, Million, etc.) was determined based on the array index.

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

The array-based implementation achieved improved performance — ~1.6 seconds over 59 unit tests — as it iterated only once over the array rather than repeatedly dividing the original number. However, performance still degraded for extremely large values.

Code Readability and Maintainability

The array logic improved readability compared to the brute-force version, but still required several temporary variables and list manipulations. The translation logic became verbose and harder to trace for future maintainers. While cleaner, it did not yet achieve the desired level of compactness and clarity.

3. Current Approach and Ditching Segmentation Approach

The final and current implementation replaces explicit segmentation with a recursive algorithm. This method leverages mathematical decomposition and recursion to translate numbers into words without maintaining intermediate arrays or lists.

This design dramatically simplified the logic — converting complex conditional trees into a compact, self-referential flow. Recursion also improved modularity, as each segment was processed by the same function call.

Additionally, currency formatting (dollars and cents) was moved out of the core function, preserving its reusability for non-currency use cases. Currency-specific logic now resides in the UI submission handler, ensuring proper separation of concerns.

Benefits

- Improved readability: Compact and easy to follow.
- Better maintainability: Minimal state handling, clear recursive flow.
- Reusability: Core logic independent from the currency context.
- Reduced complexity: No array manipulation or nested loops

3.1 Future Implementations

Although the current implementation meets project requirements, several potential extensions were identified for future iterations:

1. BigInteger Support:
Replace long with BigInteger to support numbers beyond long.MaxValue (e.g., sextillions and above). While useful for completeness, such magnitudes are rarely required in currency-based applications.

2. Fractional Precision:
Expand support for decimals beyond two places (e.g., “123.4567” → “One Hundred and Twenty-Three Dollars and Four Thousand Five Hundred and Sixty-Seven Ten-Thousandths”).

3. Localization:
Introduce multilingual support via a strategy pattern (e.g., INumberToWordsConverter) to allow translations into other languages or currency formats.

4. Asynchronous Execution:
Convert methods to async to improve scalability when integrated into a Blazor Server environment.

5. Grammar Customization:
Enhance grammatical accuracy (e.g., British vs American “and” usage).

6. Configuration and Extensibility:
Introduce settings for locale, currency, and formatting preferences through dependency injection or configuration files.

4. Conclusion

Through iterative refinement, the project evolved from a brute-force prototype into a concise, recursive, and maintainable solution aligned with modern C# best practices. Each iteration provided insights into performance, clarity, and scalability trade-offs, leading to the final approach that balances readability, efficiency, and reusability.

The project also demonstrated adaptability in learning new frameworks (Blazor) and applying object-oriented principles in a practical, test-driven context.