Approach Selection for Number to Words

## Objective

The goal of this document is to outline the reasoning behind the chosen approach for converting monetary values into words, along with the rationale for rejecting alternative solutions.

Chosen Approach

The implementation uses a combination of:

1. Decimal Parsing and Validation: Ensuring valid and bounded numeric input.
2. Separation of Concerns: Breaking the number into dollars and cents.
3. Recursive/Iterative Thousand-Grouping Conversion: Converting numbers into words by grouping into thousands.
4. Reusability of Helper Methods: Using modular functions such as [ConvertHundreds] to handle specific tasks.

This approach ensures:

* Readability: The logic is straightforward and broken into smaller, well-named methods.
* Modularity: Components like [ConvertHundreds] and [ConvertWholeNumber] can be reused independently.
* Accuracy: Proper handling of [decimal] points, rounding, and limits ensures consistent results.
* Scalability: The solution is extensible for larger numbers without requiring extensive changes.

Key Decisions

1. Decimal Parsing

Selected Approach:

* Used [decimal.TryParse] to validate the input and ensure proper error handling.

Why?

* [decimal] is ideal for currency values due to its precision.
* TryParse eliminates the risk of unhandled exceptions from invalid inputs.

Alternatives:

* Using [double]: Rejected because [double] is prone to floating-point errors, especially with monetary calculations.
* Manually Validating Strings: Rejected as it would introduce unnecessary complexity.

2. Handling Large Numbers and [Thousands]

Selected Approach:

* Used an iterative method to group numbers into thousands and convert each group separately using the [ConvertHundreds] method.

Why?

* This approach avoids recursion limits, making it suitable for extremely large numbers.
* Simplifies the logic for handling different magnitudes (thousands, millions, billions).
* Easy to extend by modifying or adding elements to the [Thousands] array.

Alternatives:

* Recursive Approach: Rejected because it may lead to stack overflow for large numbers and adds unnecessary complexity.
* Hardcoding Magnitudes: Rejected as it reduces flexibility and maintainability.

3. String Concatenation for Output

Selected Approach:

* Dynamically built the output string by concatenating dollar and cent parts with appropriate separators (e.g., "AND").

Why?

* Maintains flexibility to adapt for different currencies or localization requirements.
* Ensures no trailing or redundant spaces by consistently applying [.Trim()].

Alternatives:

* Using String Templates: Rejected because it reduces flexibility when handling optional parts (e.g., cents).
* Hardcoding Complete Sentences: Rejected as it would require significant changes for localization or format adjustments.

4. Class-Level Constants for Number Words

Selected Approach:

* Defined [Ones], [Tens], and [Thousands] as class-level constants to avoid redundancy.

Why?

* Enhances readability by centralizing common resources.
* Reduces memory overhead by avoiding repeated definitions.
* Simplifies debugging and updates.

Alternatives:

* Repeated Definitions in Methods: Rejected as it increases the likelihood of inconsistencies and maintenance difficulties.
* Global Variables: Rejected to maintain encapsulation and minimize potential side effects.

5. Error Handling and Validation

Selected Approach:

* Checked for invalid input, negative numbers, and amounts exceeding the maximum allowable value.
* Provided user-friendly error messages for each validation failure.

Why?

* Ensures robust input handling and prevents the application from producing undefined behaviour.
* Improves user experience with clear feedback.

Alternatives:

* Silent Failures: Rejected as they lead to poor user experience and debugging challenges.
* Throwing Exceptions: Rejected to avoid abrupt application crashes from user errors.

6. Precision in Decimal Operations

Selected Approach:

* Used [Math.Round] to compute cents, ensuring proper rounding to two [decimal] places.

Why?

* Eliminates floating-point precision issues that might occur during subtraction and multiplication.
* Prevents common rounding errors in financial calculations.

Alternatives:

* Direct Multiplication and Casting: Rejected due to potential rounding inaccuracies.
* String-Based Rounding: Rejected as it would unnecessarily complicate the logic.

Advantages of the Selected Approach

1. Scalable: Handles both small and large numbers without major changes.
2. Maintainable: Modular functions make it easy to debug, update, or extend the implementation.
3. Reliable: Robust error handling ensures consistent behaviour for edge cases.
4. Flexible: Extensible to support additional features like currency localization.