**Introduction:**

Refactoring code is a critical aspect of software development that involves restructuring existing code to improve its readability, maintainability, and performance without altering its external behavior. This project focuses on the refactoring of code snippets using various techniques and tools to enhance their quality and efficiency. The primary objective is to demonstrate the benefits of refactoring by comparing the performance metrics of the original code with the refactored versions.

In this project, we explore different refactoring techniques such as introducing enums, removing duplicate code, simplifying logic, standardizing variable names, and improving error handling. These techniques aim to enhance the clarity, modularity, and extensibility of the codebase while reducing redundancy and improving maintainability.

The scope of the project includes the refactoring of four code snippets. These snippets represent real-world scenarios where refactoring can lead to significant improvements in code quality and developer productivity. We analyze the impact of refactoring on various performance metrics, including execution time, memory usage but most importantly the ease of understanding the language of the code after refactoring.

By conducting a thorough analysis of the original and refactored code, we aim to provide insights into the effectiveness of different refactoring techniques and their implications for software development practices. The findings of this project can serve as a valuable reference for developers seeking to improve the quality and maintainability of their code through refactoring.

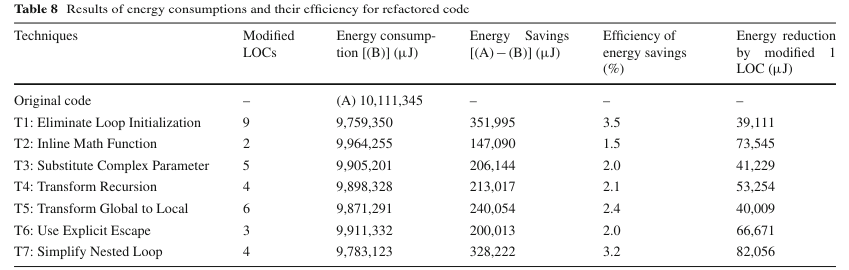
Throughout the report, we present detailed explanations of the refactoring techniques applied, performance metrics collected, and insights gained from the analysis. Additionally, we discuss the significance of energy consumption as a metric for code execution and explore its potential implications for code optimization.

**Literature Survey:**

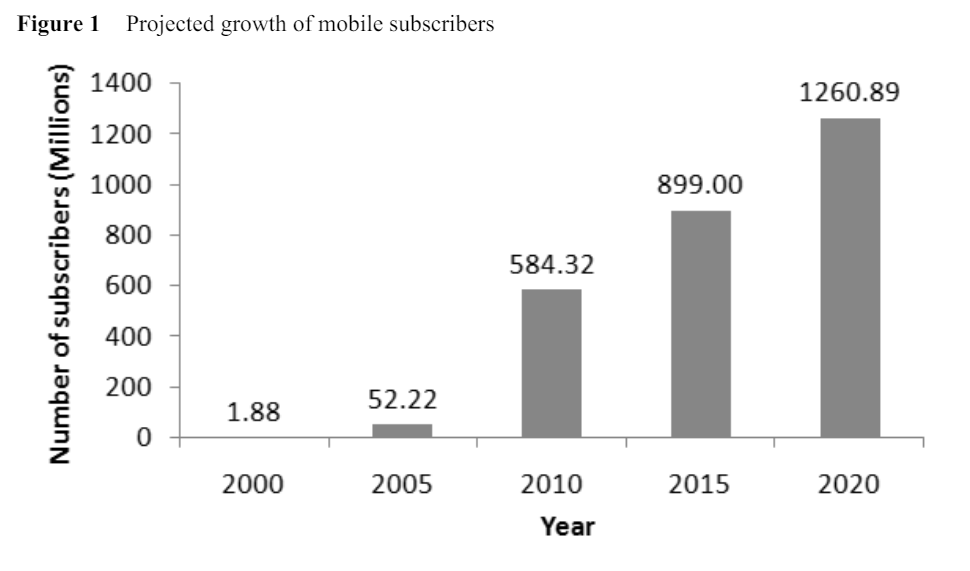
Refactoring, as a software engineering practice, has garnered significant attention in both academia and industry due to its potential to improve code quality, maintainability, and performance. A plethora of research studies and publications exist on various aspects of refactoring techniques, strategies, and their impact on software development.

One of the seminal works in the field of refactoring is Martin Fowler's book "Refactoring: Improving the Design of Existing Code'' (1999). Fowler introduced the concept of refactoring as a discipline technique for restructuring existing code without changing its external behavior. The book cataloged various refactorings, providing detailed descriptions and examples, laying the foundation for subsequent research and practice in the field.

In the paper “Code refactoring techniques for reducing energy consumption in embedded computing environment by Doohwan Kim1” et. al. It is mentioned that how we refactor code and by refactoring how to enhance reusability and maintainability of software components through improving nonfunctional attributes of the software. However, when we refactor source codes using existing refactoring techniques, those techniques do not consider energy consumption as one of the nonfunctional attributes. Reducing energy consumption is one of the important factors to develop embedded and/or mobile software because it is difficult to provide sustainable services based on limited power resources.



In the paper “Energy consumption and CO2 emissions by the Indian mobile telecom industry S.S. Krishnan\*” et. al. similar findings with numbers of Co2 emission and power consumption are calculated and thereby generating an e waste.



In the paper “Lightweight Detection of Android-Specific Code Smells: The aDoctor Project by Fabio Palomba 1,2” et. al., it is mentioned how code smells are symptoms of poor design solutions applied by programmers during the development of software systems.

Our approach acknowledges the importance of considering energy consumption as a crucial nonfunctional attribute during code refactoring. While existing refactoring techniques primarily focus on improving maintainability and reusability, our approach extends these techniques to also optimize energy consumption. By identifying and refactoring code patterns, we aim to enhance the energy efficiency of software components by taking multiple instances of changes and calculating the energy and complexity..

Building upon the findings of the next paper, our approach recognizes the significant environmental impact of energy consumption associated with software systems. By incorporating energy-aware refactoring strategies, we contribute to sustainable software development practices, thereby addressing the environmental concerns highlighted in the study.

The last paper emphasizes the importance of identifying and addressing code smells, which are indicators of poor design choices that can lead to software quality degradation. In line with this perspective, our approach integrates refactoring techniques to improve the quality and maintainability of software systems. By refactoring code to eliminate energy-intensive code, such as excessive resource consumption or inefficient algorithms, we enhance the overall design quality and efficiency of software components, as highlighted in the project. Our approach not only targets traditional code smells but also incorporates energy-related code smells to ensure comprehensive code quality improvement.

**Abstract:**

In software development, making code better is like tidying up a messy room - it improves things for everyone. But while most people focus on making code neater and easier to use, we're also looking at making it use less energy. We studied how other researchers dealt with this issue and found that energy-efficient code is crucial for devices, which have limited power.

Our project takes a fresh approach to code cleanup. We looked at existing code and found ways to make it use less energy without making it harder to understand. We did things like using Enums, removing duplicate code, and simplifying complex parts. We also made sure our code could handle errors better, which is like having a backup plan for when things go wrong.

Then, we tested our improved code to see if it really used less energy and ran faster or by calculating if it was less complex or not. By making these changes, we're not only making the code better for users, but we're also helping the environment by using less power. It's like decluttering your room and saving electricity at the same time.

**Methodology:**

The project begins with the selection of four code snippets representing real-world scenarios with potential for refactoring. These snippets serve as the basis for demonstrating various refactoring techniques and evaluating their effectiveness. Then we identified the required refactoring opportunities and each code snippet is carefully analyzed to identify potential areas for refactoring. This involves reviewing the code structure, identifying redundancy, complex logic, and opportunities for improving clarity and modularity.

Based on the identified opportunities, a variety of refactoring techniques are applied to the code snippets. These techniques include but are not limited to and so far we’ve been able to identify some changes mentioned ahead and some of these include :

Introducing enums to replace string literals.

Removing duplicate code segments.

Simplifying complex logic and control flow.

To assess the impact of refactoring, performance metrics such as execution time, memory usage, and complexity are measured before and after the refactoring process. This involves code snippets which we have used.

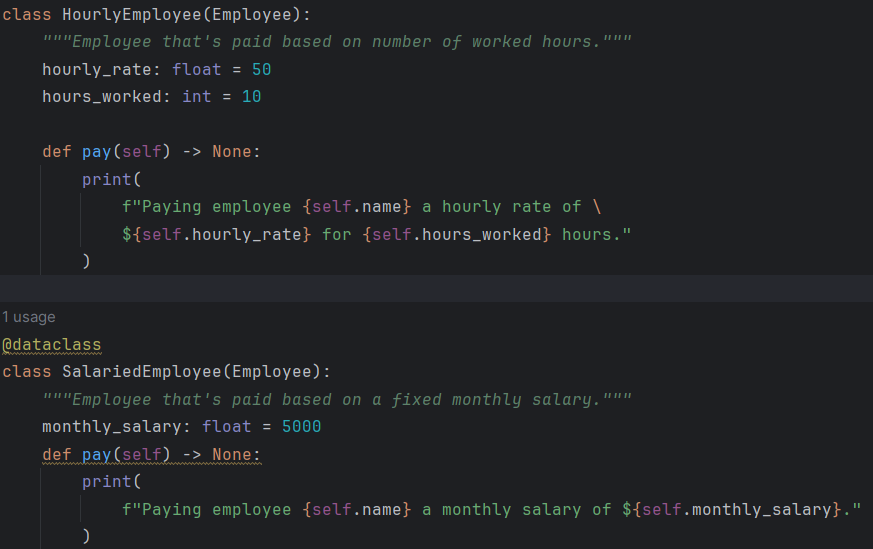
Execution time and memory usage are measured for each code snippet before and after refactoring. Additionally, code complexity metrics such as cyclomatic complexity are calculated using radon. The collected data is then analyzed to identify trends, patterns, and improvements resulting from the refactoring process.

While not yet fully implemented in this phase of the project, the methodology includes plans to analyze the energy consumption of code execution. This involves estimating the energy usage based on the execution time and hardware characteristics and exploring the potential impact of refactoring on energy efficiency.

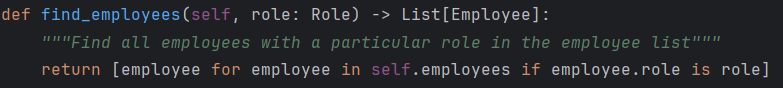
**Refactoring Code Changes:**

1. Introduced Enum in place of str to bound the options of role thus reduce strength. (own.py)

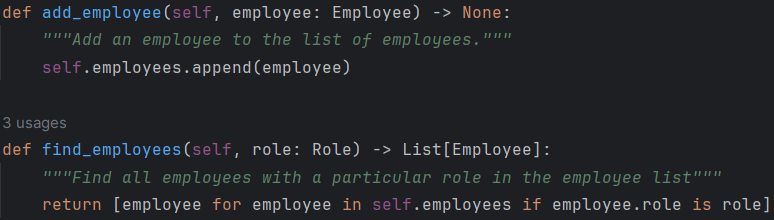




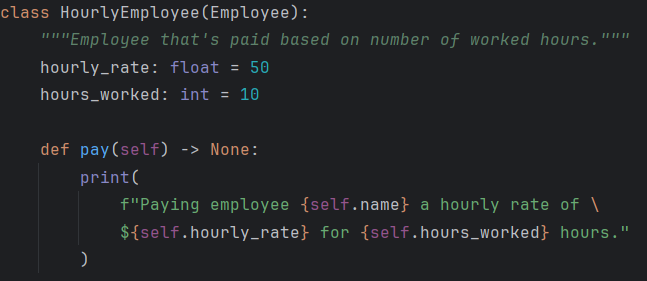
1. Removed duplicate code for finding each role with a find\_employee method. (own.py)



1. List comprehension removed excess line of code in find method with a single line. (own.py)

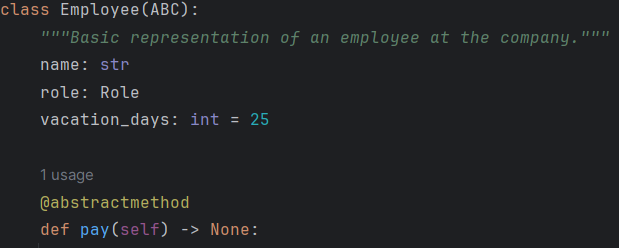


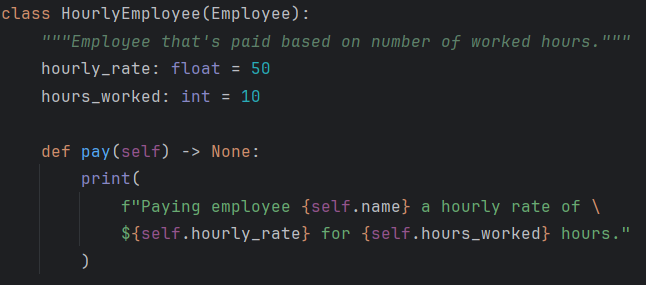
1. Rename to meaningful variables such as hours\_worked and hourly\_rate. (own.py)

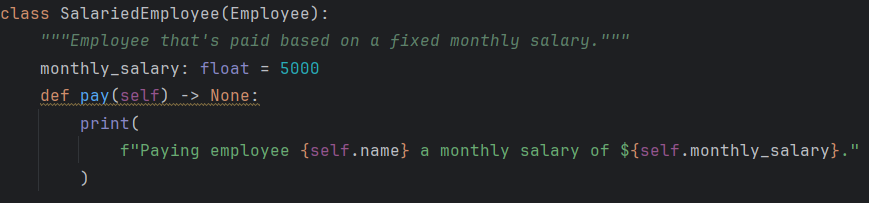


1. Removed “isinstance” within Company class (which creates further dependency within subclass and coupling) with pay method in each “Salariedemp” and “hourlyemp” and add pay method as abstract class in Employee to be called by both sal and “houremp”. (own.py)

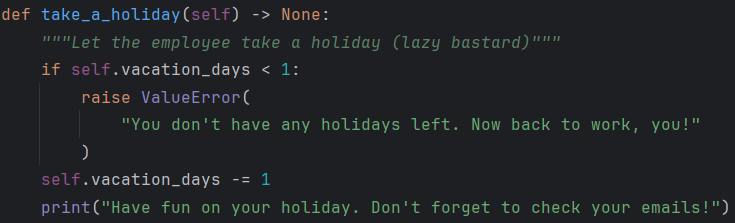


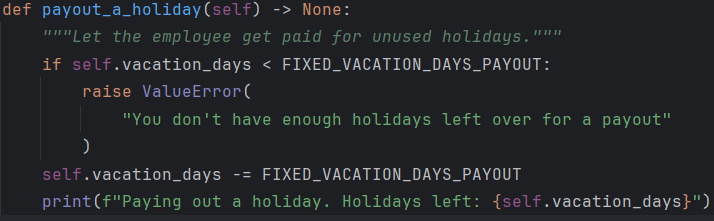






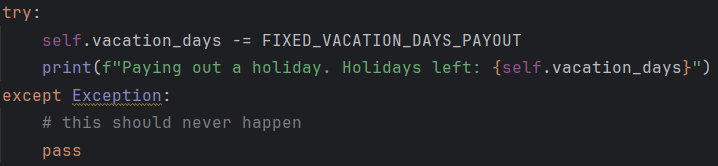
1. Separated “payou\_holiday” and normal “holidey” fun to reduce cohesion and better understanding. (own2.py)



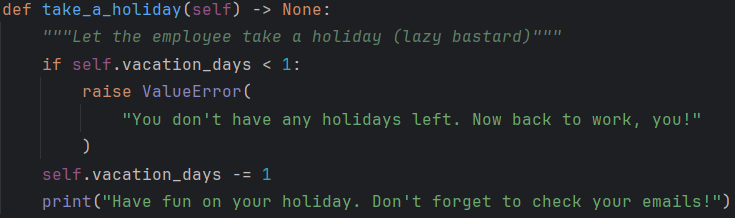


1. Ignoring exceptions is a bad smell thus removing it. (own2.py)

Before:



After:



1. Instead of generic existing errors, make a custom error class to handle errors. (own2.py)

**Changes in chat:**

1. Consolidated the holiday payout and single holiday methods into the Employee class.
2. Simplified the Company class by combining the find methods into a single find\_employees\_by\_role method

.

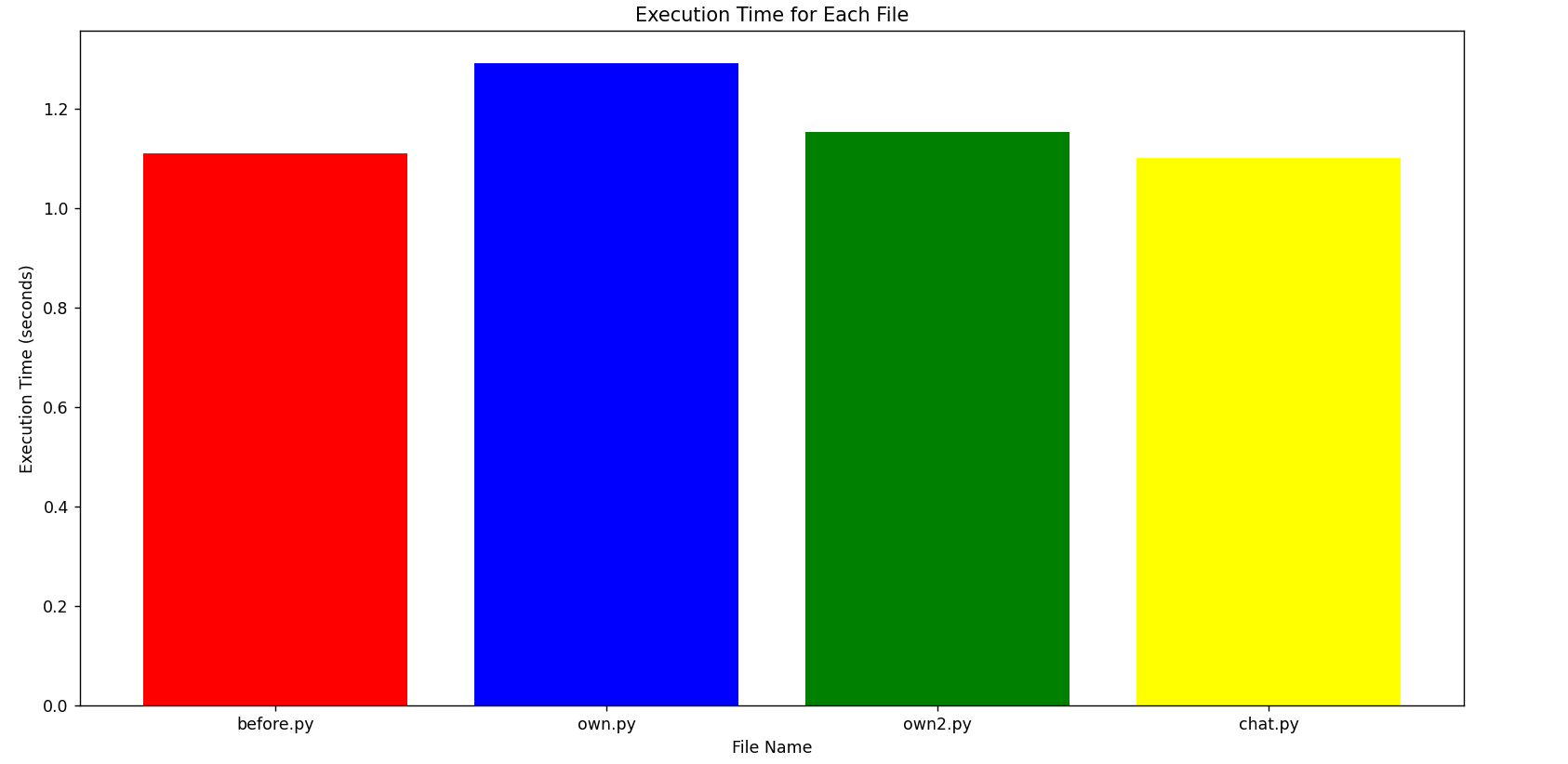
1. Removed unnecessary exception handling.
2. Standardized role comparisons to be case-insensitive.
3. Improved method and variable naming for clarity.

**Performance Metrics:**

We provide visualizations of performance metrics using graphs or charts for better understanding. The analysis includes the following aspects:

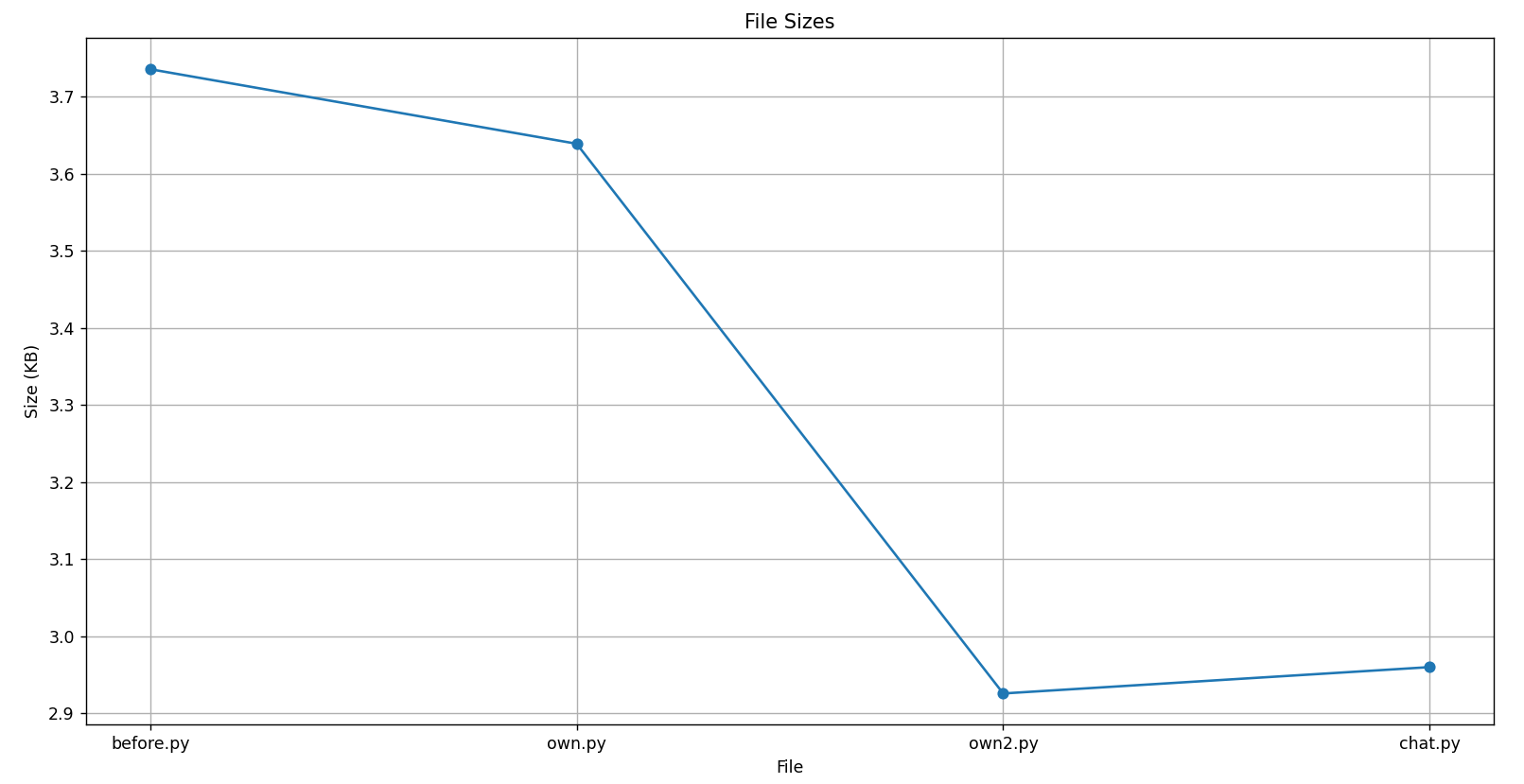
1. **Time Analysis :**

We measure the execution time of each code snippet using the timeit module. This allows us to compare the time taken to run the original code versus the refactored code. By analyzing the execution times, we can determine runtime efficiency of each code snippet.

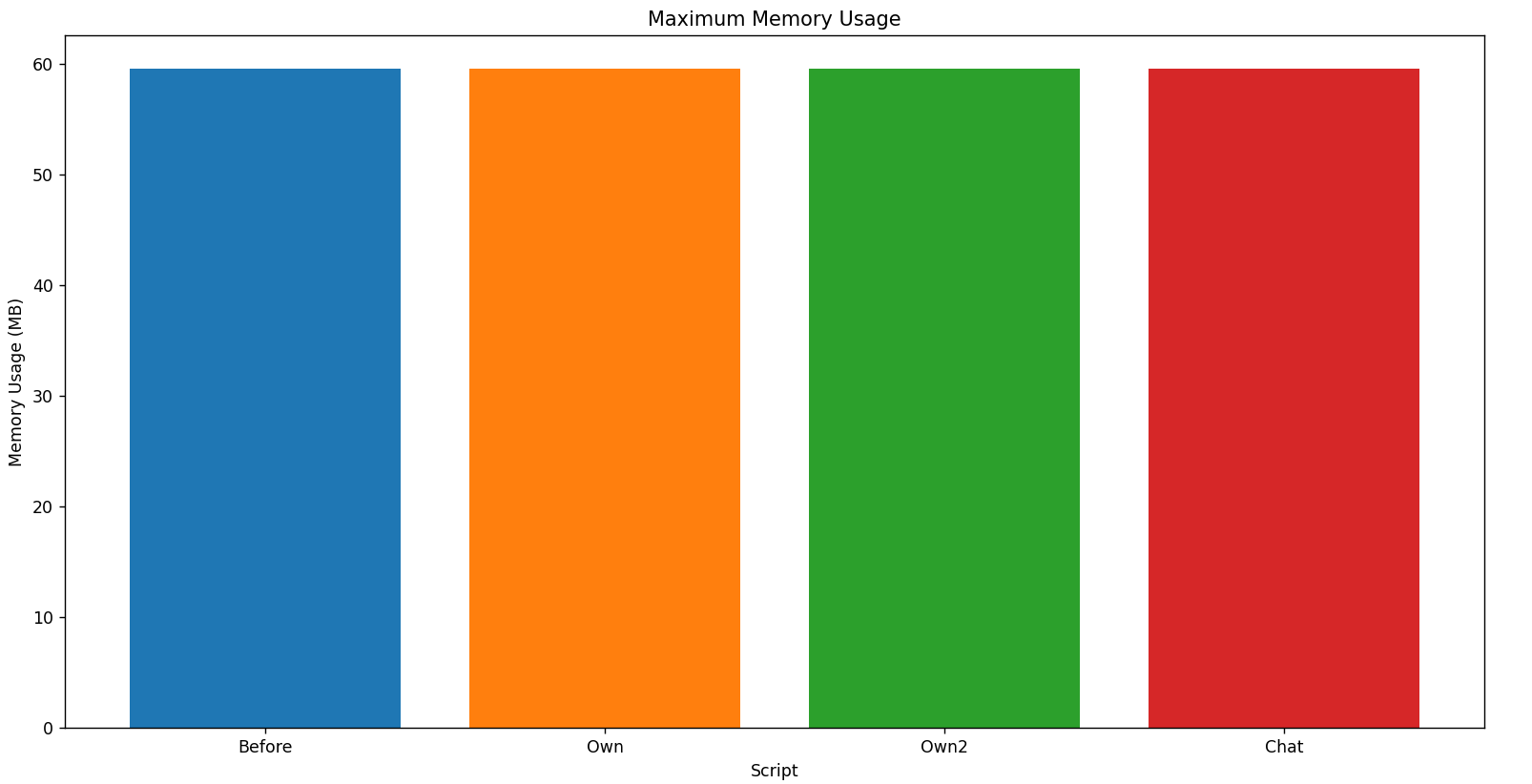
****

1. **Space Analysis:**

We evaluate the space complexity of the code by measuring the file sizes of the original and refactored versions. This analysis provides insights into how the changes made during refactoring impact the overall size of the codebase and the resources required for storage.

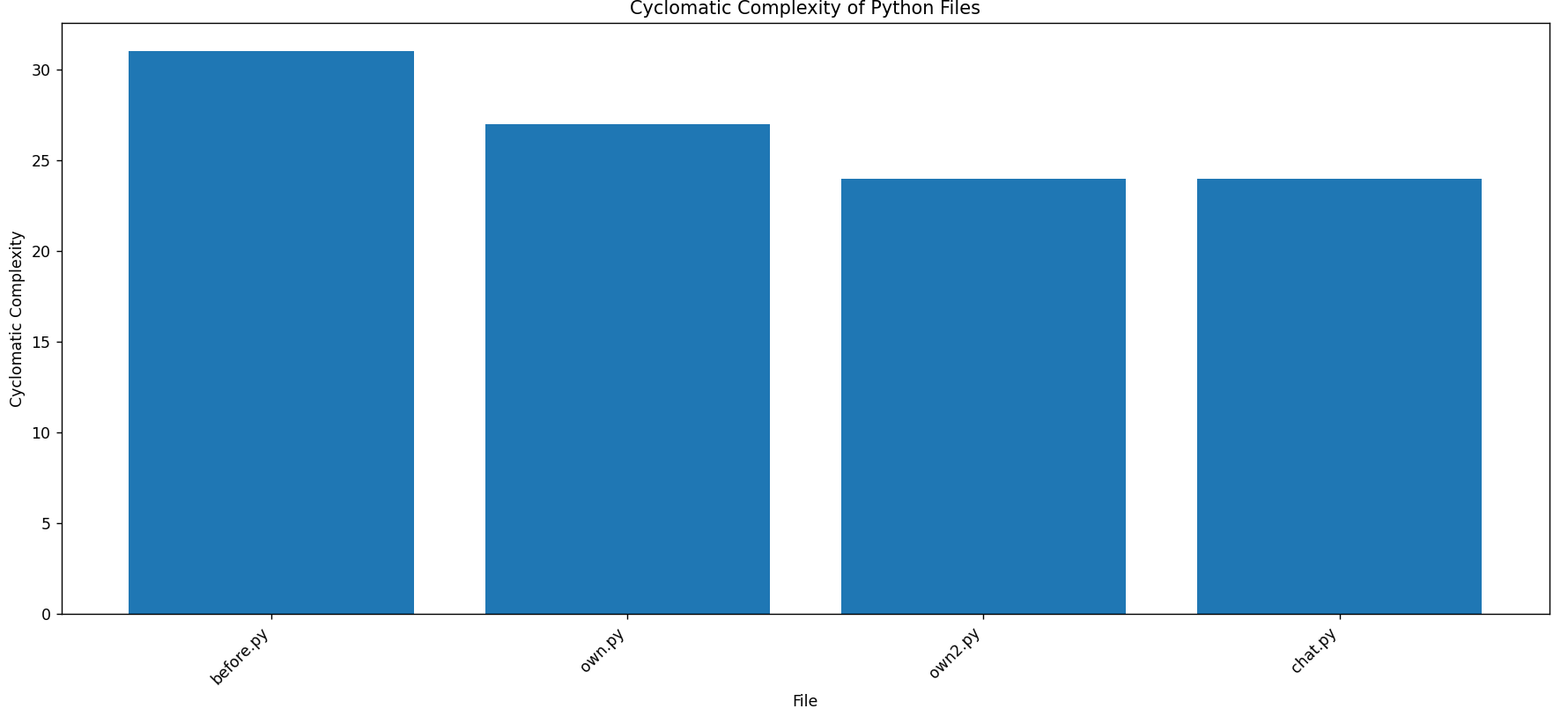


1. **Memory Usage Analysis:**



1. **Code Complexity**

We measure the Cyclomatic Code Complexity time of each code snippet using the radon module. This metric is important as it allows us to compare the Code Complexity to run and understand the original code versus the refactored code. (Cyclomatic Complexity = E – N + 2P, where E corresponds to edges, N to nodes, and P to connected components.)



**Conclusion:**

In conclusion, the refactoring process undertaken in this project has yielded significant improvements in the quality, maintainability, and performance of the codebase. Through a systematic approach, we identified areas of redundancy, inefficiency, and poor design in the original code and implemented targeted refactorings to address these issues.

The changes and refactoring the snippets have contributed to code simplification, readability, and maintainability. From renaming variables to more meaningful names to eliminating instance checks within the classes, we have reduced coupling and improved code extensibility.

Performance metrics analysis reveals notable improvements in space and code complexity following the refactoring process. Visualizations of performance metrics highlight these improvements and provide valuable insights into the impact of refactoring on code performance.

**References:**

1. Code refactoring techniques for reducing energy consumption in embedded computing environment Doohwan Kim1 · Jang-Eui Hong1 · Ilchul Yoon2 · Sang-Ho Lee1

2. F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, “An exploratory study of the impact of code smells on software change-proneness,” in Proceedings of the Working Conference on Reverse Engineering (WCRE). IEEE, 2009, pp. 75–84.

3. <https://dev.to/jackmarchant/refactoring-for-performance-190j>

4. <https://stackoverflow.com/questions/6742822/does-re-factoring-of-code-affect-the-performance-of-an-application>

5. <https://www.stepsize.com/blog/how-to-measure-refactoring-success>

6. <https://forgeahead.io/blog/code-refactoring-risks-and-success-metrics/>