Project Report

CS3060 — Final Project

Kobe Brown & Urvish Patel

December 12, 2025

Abstract

Abstract.
We are investigating how Object-Oriented Programming (OOP) in Python compares to Functional Programming (FP) in Scala when building the same application. We use the game Yahtzee as a controlled environment to test how each paradigm influences design, performance, readability, and debugging complexity. This problem matters because language paradigms shape how developers think, structure logic, and resolve errors.

(i) We designed and implemented a Yahtzee game in Python using Object-Oriented principles such as encapsulation and mutable states.

(ii) We designed and implemented a version of this Yahtzee game in Scala using functional principles such as higher-order functions, immutability, and pure transformations.

(iii) We used myriad tools to run multiple Yahtzee rounds and measured execution time, memory usage, and structural complexity.

(iv) Our findings were visualized and we critically examined our findings.

Contents

1 Introduction

Object-oriented and functional paradigms have been popular for many years, and each has a cohort of supporters and advocates. Naturally, this is because programmers tend to serve a niche, and certain paradigms lend themselves to certain advantages relative to others [1]. Furthermore, there are a unique set of disadvantages with programming paradigms as well. Because of the niche applications programmers deal with (scopre of their work/research will be narrow), programmer preference can be subjective and biased. We aim to implement the same user experience between functional Scala and object-oriented Python, and use a plethora of tools to gauge performance, readability, and debugging. The actual codebase is irrelevant, but we want a universal vision around which to build, and to have a project complex enough to analyze and uncover meaningful differences between the two. We chose Yahtzee as a moderately complex game to implement that can be realistically implemented with both paradigms. In order to shape consistent, reliable programming approaches, it is important to objectively analyze object oriented and functional paradigms[1]. We aim to uncover patterns that help us guide future programming approaches.

The subjective bias of programmers has two edges. On one, it forces parallel problem solving, with multiple paradigms and approaches being simultaneously developed, tested, and optimized[2]. On the other, much time and energy is lost in migrating between approaches (codebases). There are so many approaches that no one programmer is able to master them all, especially with the rate of change and optimization that occurs with most languages over the years[2]. We aim to implement this moderately complex program and uncover patterns so that more time and energy is not wasted in the future, and the optimal paradigm can be chosen readily.

We found that there were significant differences between memory usage and execution time between the two paradigms. The Scala functional approach proved to be 10-100x faster than the Python object oriented approach[3]. And JVM has built in optimizations for memory allocation and the JIT (just-in-time) compiler optimizes memory usage over multiple compilations. Furthermore, the differences in debugging revolve around the nature of the two languages; Scala has more nuanced and reliable error handling while Python lends itself to continuous integration with a series of runtime errors which are relatively easy to resolve. We were able to have an impressive code coverage with the Python approach, but Scala was far more comprehensive in error handling. In terms of readability, we find stark differences but struggle to quantify them. When using a naive approach, Scala technically had less lines of code overall. The immutability of the functional implementation means data isn't changing after creation, which makes the overall implementation more concise on both the software and hardware side.

2 Architecture, Design, and Implementation

2.1 Architecture

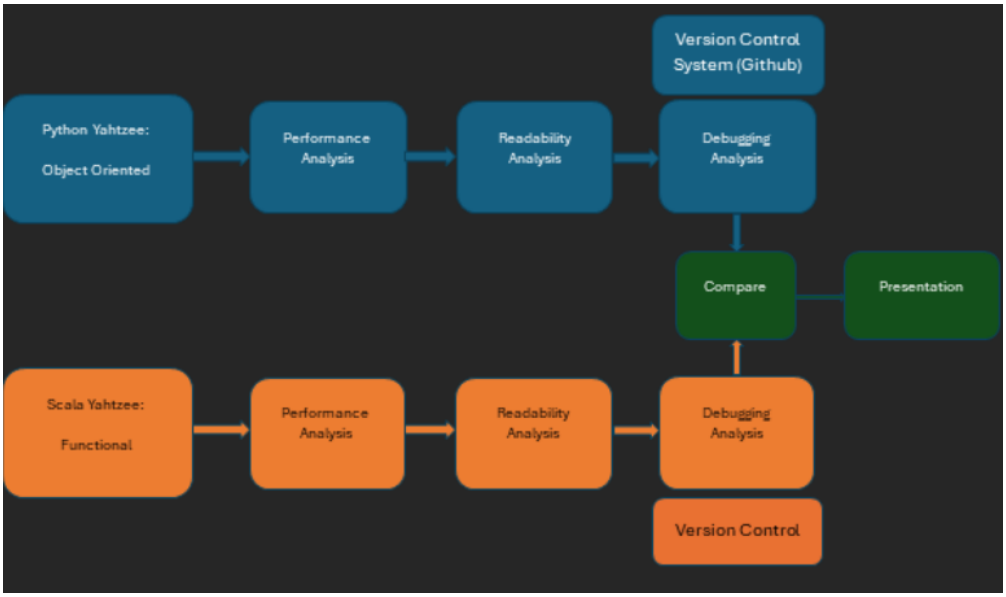

Figure 1: System architecture overview.

2.2 Design

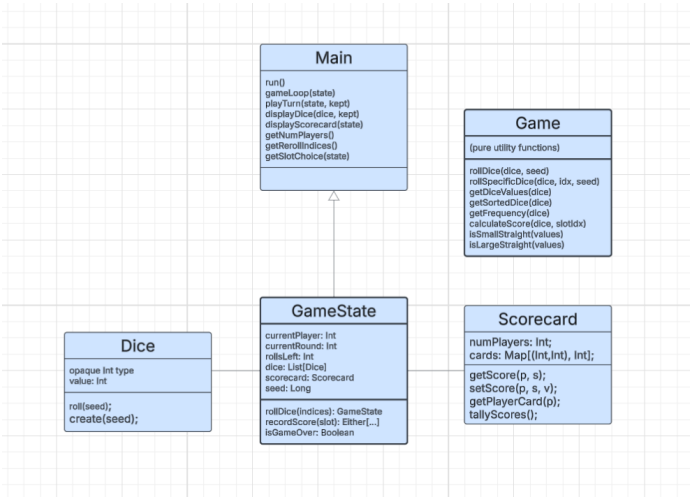

Figure 2: Scala functional design diagram.

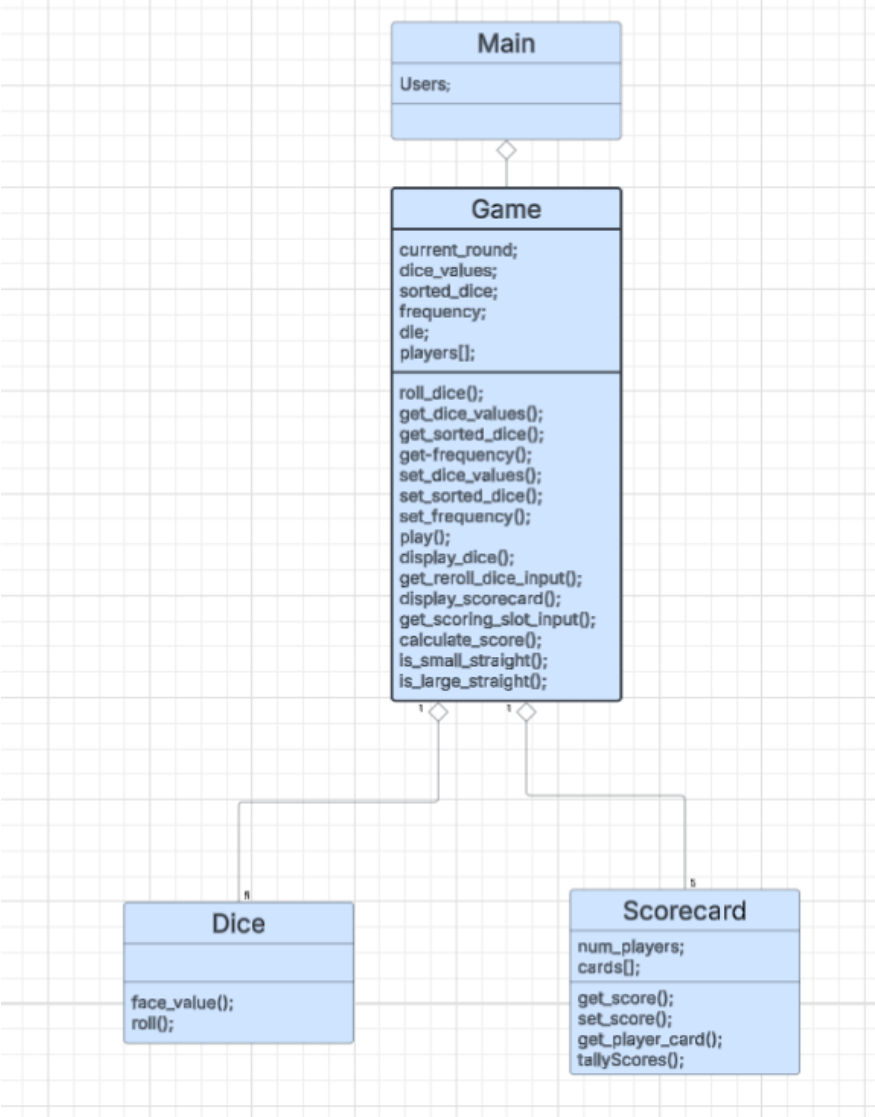

Figure 3: Python OOP design diagram.

3 Methodology

We used GitHub for Version Control to manage our repository and work remotely. We used Discord for continuous communication during development. We utilized Claude Sonnet 4.5 and GPT 4.1 as we would with our regular development workloads. The models were particularly utilized to generate tests, debugging, and code refactoring. Within the same repository, we implemented Yahtzee in Python using object-oriented principles. Then, we created a functional implementation in Scala 3. The goal was to implement both as we would with a regular development project, featuring testing, continuous integration, and debugging.

Both implementations were analyzed using niche tools. The python implementation was

analyzed using the Python Standard Library (cProfile, io, pstats). 3rd party tools like pylint, coverage, matplot, numpy, and radon (cyclomatic complexity, maintainability index) were used to analyze the performance of the python implementation. The scala implmentation was analyzed in a similar fashion. Scala has access to both the Java Standard Library (io, system, runtime, thread) and Scala Standard Library (util, collection). Furthermore, the Python Standard Libary was used as well. 3rd party tools such as sbt, ScalaTest, matplot, numpy, radon, pylint, coverage were also used. Many of the tools used overlap for both implementations.

The analyzers collected 3 types of metrics: performance, readability, and debugging. For Python, performance analysis featured measures of execution time for methods (perf counter()) and the memory used over a period of execution (tracemalloc). Measurements were made over multiple trials. The standard, default inputs are 4 trials at 21 samples per trial, because it allowed rapid analysis. Statistical analysis was applied. For readability metrics, we measured the total lines of code, with comments and blank lines filtered out. Cyclomatic complexity and maintainability were measured using radon. Finally, debugging analysis we measured test coverage and total issue count. We attempted to track the type, frequency, and location of the issue.

For Scala, performance analysis measured execution time and memory. Execution time was relatively trivial, and certain adjustments were made to measure memory usage. For the readability, we measured total lines of code, filtering for blank lines and comments. We also gauged the different type of structures (objects, case classes, def functions, extension methods, and opaque types). Ultimately, we gauged over immutability by examining val/var counts, case class count, copy usage, and map usage. For the debugging analysis, we examined test count, failure raite, error rate, skip rate, and total execution time for the tests. We examined the types of error handling used (paths through either usage).

Data collected by the analyzers is stored in json files in folder data (there is one for each implementation). That pair of stored data is used by a pair of visualizers (python scripts) to generate visual figures. These figures allow us to compare the performance of the two implementations. We collected data, analyzed the relationships, and concluded based on our findings.

4 Evaluation

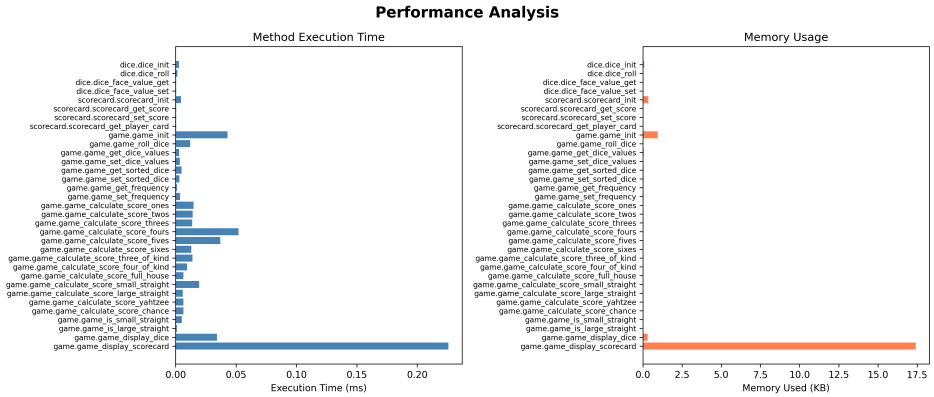

Figure 4: Python OOP performance metrics with regards to execution time and memory usage

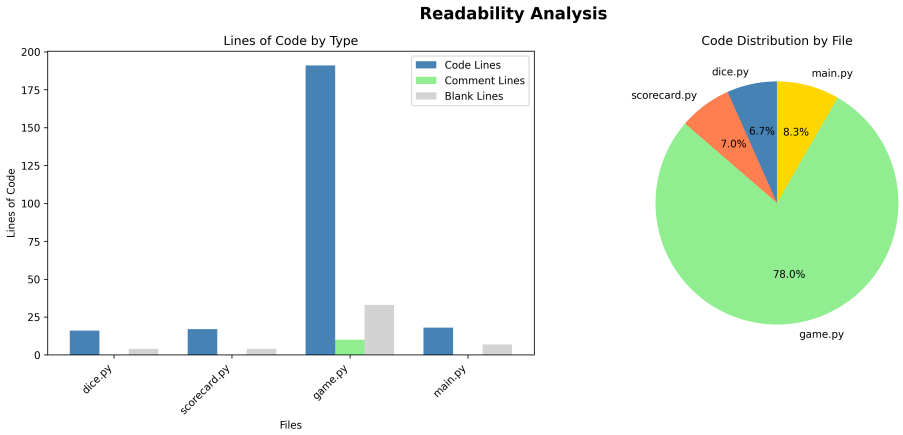

Figure 5: Python OOP readability metrics with regards to lines of code and logic distribution across classes

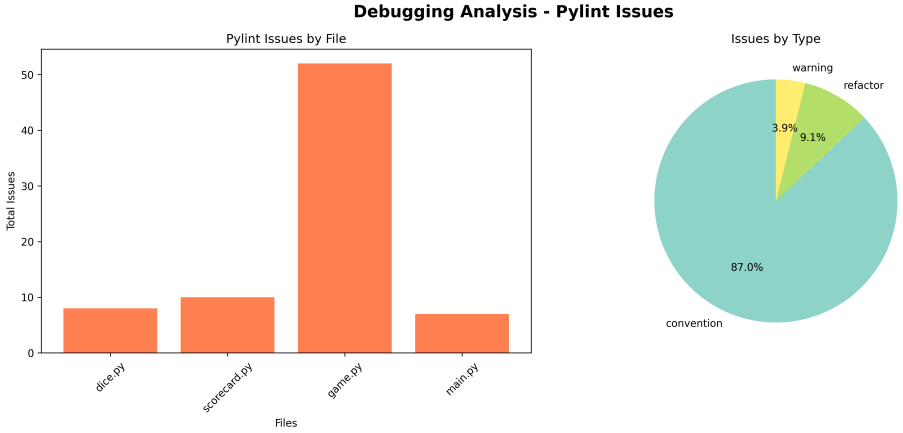

Figure 6: Python OOP debugging metrics with regards to issue distribution and type

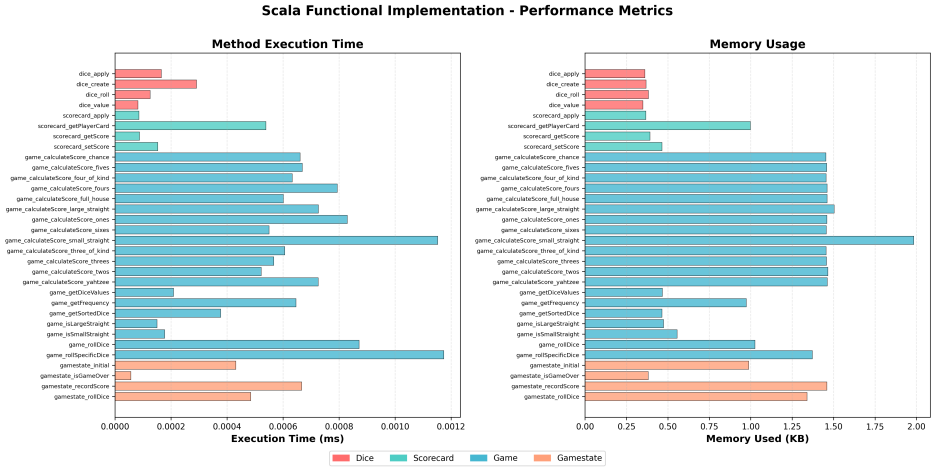

Figure 7: Scala functional performance metrics with regards to execution time and memory usage

Figure 8: Scala functional readability metrics with regards to lines of code (and distribution), structure type, immutability patterns.

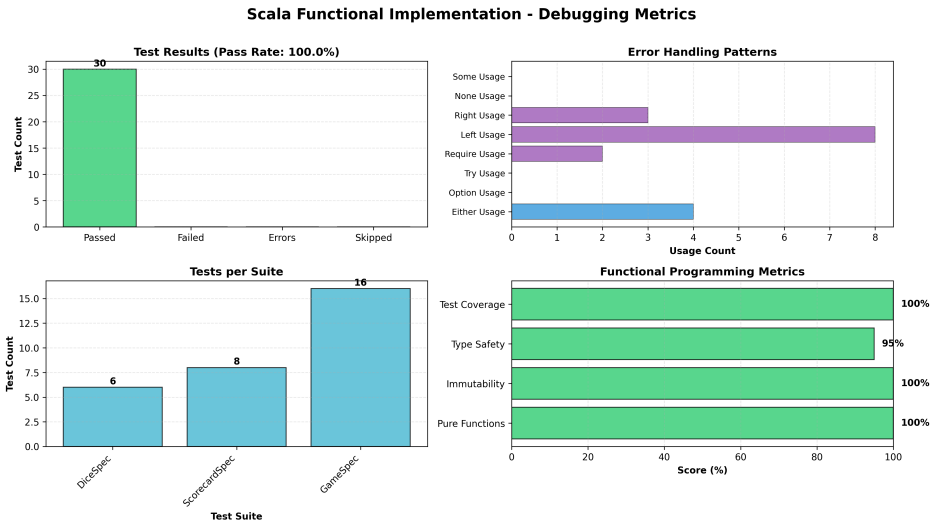

Figure 9: Scala functional debugging metrics with regards test pass rate, error handling, test distribution, and FP metrics

We successfully implemented both a Scala functional and a Python object-oriented game of Yahtzee. The game runs locally in the terminal, and operates based on user interactions/decision making. We were able to analyze both of our implementations and find a marked difference in execution time and memory usage, with an overall bias towards Scala in regards to performance (virtually same memory usage, significant improvement in execution time). We also found Scala to be more readable and better at handling errors, while Python's debugging was markedly simpler (less time spent per issue). Ultimately, we visualized our findings and consolidated our conclusions for this report.

The biggest challenge proved to be measuring the memory allocations to the heap for the Scala functional implementation. Our intial measurements found 0.00 KB memory usage for everything. We intuitively know that no memory usage at all is wrong, so we dug deeper and discovered the Java Virtual Machine's designed to optimize memory collection for objects whose lifetime is confined to a single method scope. In short, JVM does not heap-allocate, making our measurements inaccurate, because we couldn't measure the stack allocations with our tools. Furthermore, JVM does rapid garbage collection in the heap, which leads to objects being created and garbage collected before our measurement can even occur. We solved these issues by creating 1000s of objects in a batch, rendering JVM unable to stack-allocate, and producing measurable heap-allocation for our tools. This also prevents premature garbage collection. Our final measurements are not perfect, but practically useful, as we saw memory being used somewhat equally across functions, with execution time and memory usage being positively correlated.

5 Discussion

Being able to run both implementations through one GitHub repository and scripting/automating the testing, building, and analysis of the codebases was relatively simple with the use of a few external python tools. We also find it valuable that anyone can download this repository locally, and run their own localized analysis to see if the data differs based on their operating system or hardware. All the tools we used were open-source or freely available, making the work extremely accessible. In short, our findings and methodology are readily replicable.

We were limited in our time and resources, so we were not able to broaden the scope of the project to factor in multiple operating systems, different hardware set-ups, and localized data collection. Our measurements occurred within a GitHub Codespace instance. Our limited exposure/familiarity with Scala also proved challenging. However, our strengths in Java and Python helped tremendously, and we were able to adapt to Scala within a reasonable time frame.

This project provided a practical comparison of how Object-Oriented Programming and Functional Programming influence software design and developer experience. Python's Object-Oriented Programming model made it easy to break the game into familiar classes and iteratively introduce features. However, managing mutable state required careful consideration, especially when debugging unexpected behavior caused by interactions between objects. Scala's functional programming paradigm led to code that was often shorter, more predictable, and easier to reason about once the functional patterns were established. Immutability removed entire categories of bugs, but the paradigm required more upfront planning and a deeper understanding of function composition.

Valuable lessons in programming are often learned through experience. We became intimately familiar with JVM optimizations with regards to memory and even execution time. The importance of planning and design became clear as we struggled to implement a Scala functional Yahtzee game. Likewise, we recognize the benefits of continuous integration for both paradigms. Ultimately, we are confident in using functional paradigms when the execution time is paramount, and properly maximize the benefits of JVM's built in optimizations. On the other hand, we retain that the object oriented paradigm remains intuitively the easiest to implement as evidenced by our significantly faster development time and the dramatically lower issue resolution rate. Due to measurable differences between the two paradigms, we will aim to utilize the paradigms when they are most appropriate, and can foresee using both for different components of a larger project.

In general, the project highlighted significant tradeoffs between the two paradigms. We hope these insights can help developers approach their future projects.

References

[1] DataCamp, "Functional programming vs object-oriented programming." https://www.datacamp.com/tutorial/functional-programming-vs-object-oriented-programming, 2025. Accessed: 2025-12-12.

[2] Rayobyte Blog, "Scala vs python: Debugging and readability comparison." https://rayobyte.com/blog/scala-vs-python/, 2025. Accessed: 2025-12-12.

[3] Netguru Blog, "Python versus scala: Performance comparison." https://www.netguru.com/blog/python-versus-scala, 2025. Accessed: 2025-12-12.