Jakob Balkovec

CPSC 4260

Individual Project Self-reflection

Xin Zhao, Ph.D.

May 20, 2025

Current Progress

As of now, my project is essentially complete. I've built a Python-based application that detects three specific types of code smells: long methods, long parameter lists, and duplicated code (Type 1–3). The tool includes a fully functional terminal-based GUI built using the `Textual` framework. Users can upload Python files (`.py`), analyze them, view results, and save refactored versions, all from within a responsive terminal interface.

The backend performs static analysis and generates Markdown reports detailing the detected smells. For duplicated code, the tool automatically applies refactoring by extracting repeated logic into new functions. All outputs, refactored files, reports, and logs, are saved into well-organized output directories. I've also written comprehensive documentation and tests, including both unit and system tests using `pytest`. Every module, function, and class is properly documented.

In short, the tool functions end-to-end. The architecture is modular, the UI is reasonably fast (though still being optimized), and the system remains stable under load.

Challenges

The biggest challenge was detecting and refactoring duplicated code, especially Types 2 and 3. Unlike detecting long methods or parameter lists, which are relatively straightforward, duplication required non-trivial parsing, normalization, and comparison across multiple code blocks. The tool had to recognize semantic equivalence despite slight variations in syntax or identifiers, and do so without crushing performance.

Choosing the right GUI framework was another hurdle. I initially tried `Tkinter`, since I've used it before, but it struggled with responsiveness when dealing with larger files or longer analyses. Rather than diving into threading and async rabbit holes, I decided to find a more responsive, modern alternative (`Textual`).

Finally, juggling this project alongside other coursework meant time management was critical. Since I was responsible for every part , planning, backend logic, GUI, testing, and documentation, I had to manage scope and momentum carefully.

Solutions to My Challanges

To tackle duplication detection, I tokenized the source code and created a comparison system that normalized identifiers and removed unnecessary syntax. I then wrote a custom diffing mechanism to flag similar blocks, using function extraction as the primary refactoring strategy. To avoid false positives, I tuned similarity thresholds based on manual testing across various code samples. (That said, I'm still unsure whether this tuning is "allowed," so I've kept it flexible.)

To address GUI performance, I pivoted from `Tkinter` to `Textual`, a modern terminal-based

TUI framework. It proved significantly faster with large files and gave me the tools to build a non-blocking, clean interface. The event-driven architecture was intuitive once I got used to it, and the documentation helped fill in the gaps.

For time management, I broke the project into daily milestones and used Git extensively to track progress and avoid regressions. Working in small, testable increments helped reduce bugs, and writing tests early meant I could refactor confidently later on.

Programming Langauge

I chose Python as the primary language for this project. Its built-in modules for AST manipulation (ast, tokenize), string processing, and functional programming made it ideal for static analysis and transformation tasks. Its readability and flexibility let me focus on implementing core functionality without spending time on boilerplate or setup.

That said, Python's reliance on indentation for block structure introduced some headaches. Since I was reading code blocks as raw strings and manipulating them, maintaining indentation became a real problem. My solution was to add a parameter to each line to track indentation, basically a "Hello Kitty band-aid on a gunshot wound." It worked, but at the cost of extra overhead during every operation.

In the end, Python made the tool possible. Despite some quirks, it allowed me to write clean, modular, testable code while supporting all the features I needed.

GitHub

If you'd like to check in on the project's progress, it's available on my `GitHub` profile. Feel free to look around. If you start with the `README.md`, everything should be pretty straightforward.

Link: `https://github.com/j-balkovec/CPSC4260`