C-Factor

Joshua R Shewmaker
Disciplined Development
University of Michigan-Dearborn
Dearborn, Michigan
jshewmak@umich.edu

Ashley M Baker
Disciplined Development
University of Michigan-Dearborn
Dearborn, Michigan
ashleybm@umich.edu

Benjamin J McAllister
Disciplined Development
University of Michigan-Dearborn
Dearborn, Michigan
bjmcalli@umich.edu

Constantin D Balan
Disciplined Development
University of Michigan-Dearborn
Dearborn, Michigan
balan@umich.edu

***Abstract*— C-Factor is a tool used to identify problematic code design within a C++ project. C-Factor uses an easy to use UI to display code smells in cpp files uploaded by the user. With the tool's 75.5% accuracy, it can find patterns in poor coding habits to improve your team's overall coding quality. Github Repository: bit.ly/CodeSmellVis**

I. Introduction

C-Factor is a C++ refactoring tool designed by Discipline Development. C-Factor is an easy to use refactoring tool for project managers and developers as well as a teaching tool for new programmers. C-Factor can read in any cpp file you have on a local drive using its simple file picker solution. This feature only allows cpp files to be selected to eliminate any confusion when looking at large projects with multiple file types. Once a file is selected C-Factor will process the code and search for the following code smells: Large Methods, Long Parameter Lists, Duplicate Code, Large Classes, and Lack of Comments. Once the code has been processed, the output will be displayed in an easy to read table format, sorted by code smell. This tool as a 75.5% accuracy for identifying problematic code. Each code smell will carry with it the line number it begins on for easy identification, as well as a brief description of why the code may be problematic. For easy comparison of amount of each code smell, C-Factor also comes with a graph for easy visualization of amount of each code smell, to allow the user to notice any problematic programming behavior.

C-Factor was written exclusively in Python 3.x. C-Factor use Tkinter for its graphical user interface, as well as MatPlotLib for C-Factor's graphing feature.

II. Problem Statement

A. Research into Problem

Discussion into the decision of the project was inspired by conversations with our Professor Marouane Kessentini about the content of the course. Kessentini lamented that such tools were his goal for the projects selected in class. At our selection of project, there was very small lecture material covered. However, the discussion of technical debt spurred our discussion further.

Two of our group members have corporate internship experience. Joshua, has Project Management experience, which influenced his need for a tool to demonstrate the validity and effect that refactoring can have on software. Joshua's hope was to use this tool to demonstrate these effects to other project managers in an easy to use interface. Simplicity was key for Joshua in the design of this project.

Ashley has a similar experience in running software projects as lead developer, quality assurance, and project manager. This provided a unique position to have three types of software roles represented through the team. In the experience drawn from these roles, a tool in this realm was the best course of action.

B. Selection of Problem Statement

With this experience, the team discussed a few options.

1) A tool that recorded and visualized Developer productivity through code smell creation.

2) A tool that identified code smells based on metrics defined by the team.

3) A tool that visualized code smells as red areas in a building diagram.

After identifying these tool ideas, the team thinned them through a few avenues. The developer productivity tool was identified as a way to invalidate morale. Due to this, we discounted this tool opportunity early. After review of the other ideas, we identified that the building diagrams were out of reach due to our experience with python. Version 1 does not include a building visualization of the code smells. Version 2, would more than likely contain this feature.

After analyzing these tool ideas, the team pilled reports of technical debt together. According to a 2018 report created by Stripe, "~300 billion global GDP loss from developer inefficiency annually." This makes tools that analyze technical debt and suggest refactoring very valuable in their exchange rate.

The final tool suggestion was selected as, a tool that identified code smells based on metrics defined by the team. From this suggestion, we developed the following problem statement: lack of a simple educational tool for C++ to demonstrate refactoring, for educators and project managers.

C. Goals and Metrics for the Problem Statement

At this stage the team defined metrics for the Code Smells. The following code smell metrics were defined.

1) Large Methods: Methods that are 50% larger than the average method length.

2) Long Parameter List: Methods that contain 5 or more parameters.

3) Duplicate Code: Code segments that contain the same text for 5 or more lines.

4) Large Class: Classes the contain 25% more methods/variables from the average count.

5) Lack of Comments: Identify classes or methods that do not have comments.

Defining these metrics was the first step in creating our program. The team deliberated over these in our first kick-off meeting. With these definitions, we will rate a 60% accuracy after developer review of the code smell identification.

III. Methodology

A. Team Management

After the first kick-off meeting, the team had weekly meetings on Sundays in the morning. These meetings were meant as Stand-Up meetings mirrored in an agile software creation methodology. As in the Agile Development Cycle, our team identified the requirements for our cycle, planned the release, designed the features, developed the application, released the code, and monitored for any bugs between releases.

B. Description of Solution to Problem Statement

The problem statement has a few elements to be examined. Educational tool is a key point of the problem statement. Expanding this statement, a few qualities to educational for the team were: "easy to understand", "simple", "relatable", and "visual." When applied to the users themselves, project managers and educators, the assumption is made that they would have a basic understanding of C++.

C. Selection of Tools

The team's goal in the methodology of the project was to utilize existing tools to reduce our footprint of work. In this, each individual installed a few Python tools to aid in the development of the application.

SonarLint was one such tool. SonarLint is a pre-emptive style and software refactoring tool that indicated the creation of code smells or style errors. This was suggested by Benjamin as a way to identify issues in our development as the team continued.

Tkinter, was a python package used to display the interface. It was selected due to its ability to select files from a given interface.

MatPlotLib, was the chosen tool for graph visualization. These were used to create the bar chart within the application.

PyUnit, a popular unit testing library for python was the unit testing solution of choice. After identifying the low-level methods, we created unit tests for these methods.

D. Resource Appropriation

At the start of the project, there were features identified for the first cycle. These consisted of backend development. Therefore, Benjamin took lead on this first cycle with Ashley as a pair programmer (XP), while Joshua and Constantin design and researched GUI interfaces. Before Constantin and Joshua began, the team agreed on a possible solution to the GUI mockup shown in Fig 1. This distribution of resources, was extremely effective for the deadline. The first cycle's features were the code parse and segmenting tool. In planning of this first cycle, a class diagram was created.

The second cycle included the code smell interpretation as well as a GUI interface with a graph. In this, Constantin and Joshua created the database together, while Benjamin created the code smell interpretations. Ashley researched Unit Testing for the project during this cycle.

Finally, the GUI and Backend components were married by Joshua, Ben, and Constantin in the third cycle. Ashley wrote low-level unit tests for a few sections of the code. These led to the results and a new class diagram including all piece of the project.

IV. Result

A. Description of Code Base

The program consists of 4 different Python files which each have their own task. The "Input.py" package launches the GUI, lets a user pick a file, and displays the output.

The "CodeParse.py" package takes a path to a C++ file and splits the code into a list of segments. These identify what kind of code it is (Literal value, word, symbol, header).

Next, the "DataStructure.py" package parses through these segments and create a tree of classes, methods, loops, conditionals, and expressions. This is demonstrated in Fig. 2.

Finally, the "SmellIdentifier.py" package searches the tree for the code smells we are searching for. It is possible for some of these functions could be merged to improve performance, but for readability the team decided to split them up.

B. Functionality Results

The software application can identify five different code smells, from a single file cpp application. Each code smell is adaptive to the style of its programmers due to the average length class and method functionality. Lack of comments is triggered when a function does not have comments preceding in a manner of pre/post conditions.

The resulting GUI as displayed in Fig. 3. is easy to understand and intuitive to new users. It displays the location, description of code smell, as well as possible code smells.

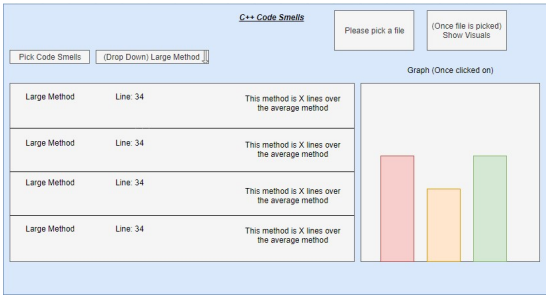


Fig. 1. Initial GUI Mockup for the Application

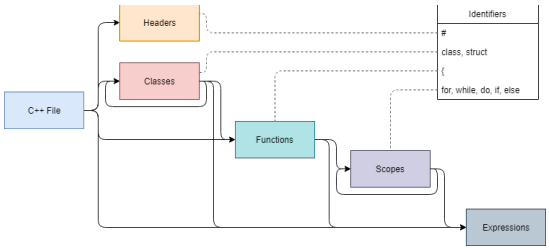


Fig. 2. Diagram of Expected DataStructure.py Parse

C. *Unit Testing Results*

Two unit testing files were created, with a total of five tests within. They are simple proof of concepts for the rest of the files. Moving forward, the team would implement these as development occurs. These results are displayed in Fig. 4.

D. *Refactoring Results*

Through use of SonarLint the team identified and fixed code smells as time elapsed. This greatly improved not only the detection of code smells, but the time it took to correct them. Since the code smells were generally discovered within a few minutes of creating them, the technical debt was quickly fixed without large scale refactoring needed.

An example code smell identified by SonarLint was the "create_method" function. It registered the complexity of this method to be 48, when the max complexity was 15. The team decided to let this code smell exist so that we could examine a dedicated Python refactoring tool.

For the final code quality testing we passed the code through **SonarQube** which identified code smells already identified by SonarLint. SonarQube suggested the amount development time (technical debt) that was needed to fix the code. With two code smells identified by SonarQube, it estimated one hour of technical debt. These code smells were complex methods that needed to be split into smaller pieces. These methods were too complex to be in a standard variable format.

E. *Accuracy Results*

The program's accuracy was 75% accurate in 106 identified code smells. 4.7% were identified as a flaw in logic the team was aware of recorded. 19.8% were considered not accurate by the standards the team enforced. Currently, the main is checked for pre/post conditions which is inaccurate. This is a simple fix the team could take care of. These results are displayed in Fig. 5., with an emphasis

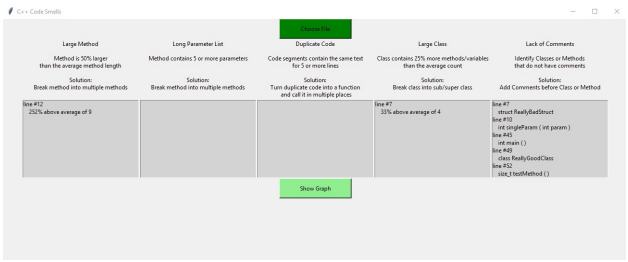


Fig. 3. Resulting Graphical User Interface (GUI)

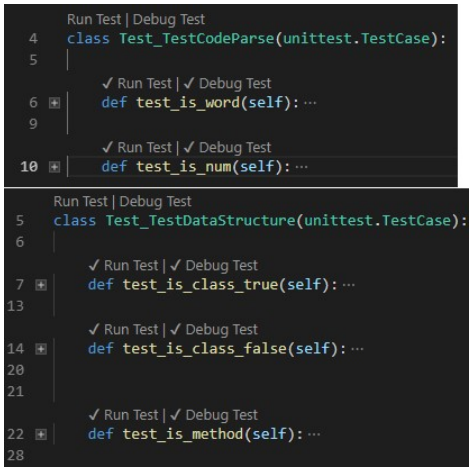


Fig. 4. Unit Testing Results

on the different types of code smells between developers.

Another developer on the team suggests that there be a minimum line for classes and methods to prevent code smell flags when the average is extremely low.

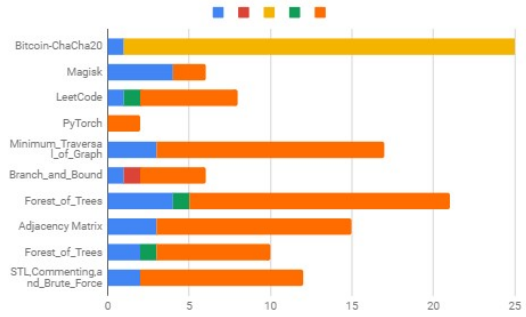
V. Threats to Validity

A. *Challenges/Limitations*

The project was not adopted from another class, so the idea and implementation were all done during the semester. It was done in python and none of the group members were thoroughly familiar with the language. This restricted how efficient and complex the program could be.

Discovering tools such as Tkinter, matplotlib, PyUnit, and SonarQube were extremely useful; however, the unfamiliarity with them led to large amounts of time used for understanding how each of them functioned. Additionally, this meant the tools could not be used to their full potential, which in turn disallowed for the program to reach the original scope.

The final limitation to the project was refactoring being a concept we were understanding and applying in the class. It being a complex and currently researched field, there was only so much that could be properly used in the project.



*Bitcoin-ChaCha20 exceeds graph, resulting amount: 297

Fig. 5. Code Smells Identified per Project

B. Shortcomings

Due to knowledge and time restrictions, the project had features cut from the final build leaving it with what was presented in class. One of them being compatibility only with C++ files. Multiple file type inputs could have easily been implemented, but for the sake of simplicity it was cut down to only C++.

Another shortcoming that has to do with the lack of understanding with refactoring was the set constraints for the code smells. This only allowed for five specific types of code smells: large methods, long parameter lists, duplicate code, large classes, and lack of comments. Each had different parameters for code that qualified as a code smell.

The last foreseen shortcoming was the requirement of python on the machine the program was running on. It was acknowledged by all group members that a python program wouldn't necessarily immediately work for everyone either due to having an outdated version of python or lacking it all together. However, it was eliminated as a worry because of the assumption that the installation/updating of python is trivial for those in need of file refactoring.

VI. Conclusion

A. Summary

Refactoring is an overlooked part of software engineering that plays a crucial part in keeping legacy software running for extended amounts of time without bugs. We focused on using refactoring to search through user input C++ files. These were displayed to the user as a graph representing the code smells, along with where in the program they were located.

B. Significant Results

The results we received from running multiple programs through C-Factor were successful. It passed the 60% that was defined in the Methodology and met expectations that were originally conceived. The identification of code smells in the programs that were run, with assumptions in place, were helpful in improving the performance and readability of the programs.

C. Future Plans

C-Factor could be immensely improved on in the future with a multitude of additions that would both improve backend functionality, and user experience. Additional code smell identification will be the priority upon improving the program to further improve the programs of the users. Some code smells that are already in the works are: excessive use of literals, contrived complexity, and data clump.

More visualization tools are also planned to be added in the future. Once the Tkinter package is fully understood by our engineers(lol), different graphics such as a pie chart or a city type display will be shown to the user to visually aid them in understanding where to improve their program. Additionally, multi-file projects are a desired feature to be added to the program. This would allow the user to upload multiple files (either of C++ or another language) to the program at once, where they would be refactored simultaneously and output. The last feature that is planned to be added is the solution to the code smells. Instead of only displaying the discovered smells, the user would also be given info on what should be done to remove said smell from their own program.

Another solution to expand automated testing would be to integrate a tool such as Selenium to test and verify the user interface and expected output through point and click testing.

References

[1] Wu, Magisk: A Magic Mask to Alter Android System Systemless-ly. https://github.com/topjohnwu/Magisk/blob/de0064af477179e7168c6bdfa07d224114e87d85/native/jni/su/pts.cpp, 2018.

[2] Liuyubobobo, LeetCodeAnimation: Demonstrate all the questions on LeetCode in the form of animation. https://github.com/MisterBooo/LeetCodeAnimation/blob/921cc799db55c4a73708ed51e1514c468c339d66/0024-Swap-Nodes-in-Pairs/cpp-0024/main.cpp, 2018.

[3] A. Suhan, PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://github.com/pytorch/xla/blob/655011d53a105e43d7fa2822446bf36c49a63830/torch_xla/csrc/passes/remove_unused_forward_outputs.cpp, 2018.

[4] Stripe, "The Developer Coefficient", Stripe, 2018 [Online]. Available: https://stripe.com/files/reports/the-developer-coefficient.pdf. [Accessed: 02- Dec- 2018]

[5] J. Sonmez, "Best Programming Language to Learn: The Top 10 Programming Languages To Learn In 2018 - Simple Programmer", Simple Programmer, 2017. [Online]. Available: https://simpleprogrammer.com/top-10-programming-languages-learn-2018-javascript-c-python/. [Accessed: 02- Dec- 2018]

[6] P. Wuille, S. Dobson, R. Ofsky and D. Bernstein, bitcoin. https://github.com/bitcoin/bitcoin/blob/452bb90c718da18a79bfad50ff9b7d1c8f1b4aa3/src/crypto/chacha20.cpp: MIT, 2018.

[7] J. Shewmaker, Minimum Traversal of Graph. https://github.com/Jshewmaker/Minimum_Traversal_of_Graph, 2018.

[8] J. Shewmaker, Branch and Bound. hhttps://github.com/Jshewmaker/Branch_and_Bound, 2018.

[9] J. Shewmaker, Forest of Trees. https://github.com/Jshewmaker/Forest_of_Tree, 2018.