CIS400/401 CodeScore

Dept. of CIS - Senior Design 2013-2014*

Allison Pearce
alpearce@upenn.edu
Univ. of Penn
Philadelphia, PA

Spencer Lee
lesp@upenn.edu
Univ. of Penn
Philadelphia, PA

Tanvir Ahmed
tanvir@upenn.edu
Univ. of Penn
Philadelphia, PA

Will McDermid
wmcd@upenn.edu
Univ. of Penn
Philadelphia, PA

ABSTRACT

We propose an application to quantitatively analyze code based on the presence of code smells and algorithmic complexity. Our application will then compute a CodeScore, which is a single metric whose value will immediately let someone determine that piece of code's quality. Our primary motivation for this project is to enable people to not only improve their own coding ability by getting feedback on their code but to allow them to objectively determine whether a piece of software is good. Similar applications have been built in the past, but they largely consist of techniques which determine whether code smells are present in code. Our application aims to not only do this, but also learn which code smells are more harmful than others and come up with a single metric that determines the overall quality of a piece of software.

1. INTRODUCTION

The application that we are proposing will quantitatively analyze software to determine its code quality. To this end, our application will test for the presence of code smells, analyze code complexity, and compute a final score that will help determine how good the software is.

Code smells are defined as symptoms in code that indicate deeper problems in the software. Some common code smells include:

- Classes that have too many lines of code
- Methods that have too many lines of code
- Duplicated code across the same piece of software
- General lack of comments when they are needed
- Long list of parameters for functions
- Conditional statements in the code are too complex
- Unhelpful naming conventions

The presence of code smells is typically correlated with low code quality since they directly affect the software's maintainability and readability. Although some work has been done in this area to detect the presence of code smells in software, what we are proposing with regards to code smells is unique. We are proposing a mechanism and algorithm that will learn which code smells are considered to be more detrimental than others and devise a way to assign weights to each kind of code smell.

For code complexity analysis, we will devise a way to see how arbitrary pieces of code perform when provided large inputs. This will allow us to test two key metrics: scalability and complexity. If the code can handle large inputs with relatively logarithmic time complexity, this will ensure that the code is likely the most optimal that it can be as well as scalable. Both of these characteristics are considered to be valuable when evaluating code.

We think that such an application will be useful in a number of fields. It can help developers improve their own code once they know which facets to improve on. Given that our application can determine what makes a piece of software bad, it can also give tips and hints as to what can be done to improve. Another use case would be for recruiting technical talent. Often, technical recruiters glance at resumes and determine whether or not to proceed with a candidate. Our application will allow recruiters to effectively determine which candidates have the strongest coding ability. This will help them save time and money by not interviewing the wrong candidates. For Computer Science teachers, the application provides an additional metric that could be used to grade assignments. Not only can teachers now check code for accuracy in functionality, they can also evaluate their students on the quality of their code. This will help students become better developers.

2. RELATED WORK

In this section, we will provide background information on the different methods of analyzing code, the standards for measuring code quality, and various services and applications that already strive to provided quantitative analyses of code quality.

2.1 Code Analysis Methodologies

The majority of existing applications for measuring code quality rely on well-established code analysis techniques. These techniques involve breaking code down into specific units and measuring the counts of these units. Such techniques include parsing the source code into control structures [9], tokens [7], assembly instructions [15], or objects [4]. Once the source code has been parsed into some unit (this process is called static analysis), the attributes of the code, such as reliability, security, efficiency, size, and maintainability can be measured from the parsed results. The actual approach to measuring these attributes originated in [2] and later became part of the ISO/IEC 25000 series of

*Advisor: Chris Murphy (cdmurphy@cis.upenn.edu).

standards relating to the quality and evaluation of software [8]. These techniques and standards comprise the foundation for today's code quality measurement applications, and they will be the basis for our approach as well.

2.2 Ohloh

Launched in January of 2006, Ohloh [1] is web service and online community owned by Black Duck Software which provides basic metrics for over 650,000 open source projects containing over 24,000,000,000 lines of code located in over 620,000 source code repositories and managed by over 3,000,000 contributors. These projects are retrieved from source control repositories and analyzed. Metrics such as lines of code, amount of source code comments, and estimated software cost (based on the COCOMO model [3]) are provided, along with commit statistics, project activity, and programming language detection. This data is graphically displayed when one views a projects information on the site. Ohloh also provides global statistics across all projects for different programming languages and contributor statistics for different authors of open source code.

Ohloh's primarily focuses on tracking project/contributor activity for large open-source projects, with less attention being paid to actually providing quantitative measurements of code quality.

2.3 Common Weakness Enumeration

The Common Weakness Enumeration (CWE) [11] is a community-developed list of software weaknesses hosted by the MITRE Corporation. The CWE was developed with the intention of providing:

- a common standard of identifying, mitigating, and preventing software weaknesses.
- a common source of source of measuring weaknesses for software security tools.
- a common language for describing the various types of software weaknesses that exist in computer architecture, software design, and source code.

CWE supports the discovery of common types of software security flaws such as buffer overflows, handler errors, pathname traversals, and resource management errors (amongst others) within code.

CWE began with the Common Vulnerabilities and Exposures (CVE) list in 1999 [10]. As part of the National Institute of Technology's (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) project [13], MITRE expanded upon the CVE list with the Preliminary List of Vulnerability Examples for Researchers (PLOVER) [5]. PLOVER was the first attempt to take real-world examples of software vulnerabilities and abstract them into common classes of more general vulnerabilities that can arise during the software development process. The goal of PLOVER was to make this information available to a wide variety of people so that they may use it for a variety of purposes. CWE encompasses much of the CVE list and expands upon PLOVER by establishing community-developed definitions and descriptions of these common weaknesses.

CWE, though possibly a valuable resource for our endeavors, primarily strives to provide standards relating to the weaknesses of code in terms of security.

2.4 Other Static Code Analysis Tool

A wide variety of other static code analysis tools also exist. The Web Application Security Consortium provides community-sourced list of some recent tools for code analysis [6]. Some of these tools include:

- CodeSonar - A full-featured code analysis tool with binary analysis, results visualization, and a wide variety of metrics. CodeSonar can detect numerous code faults and security risk within projects containing up to millions of lines of Java, C, or C++ code.
- Klocwork Insight - Provides a wide range of metrics with detailed reports and offers integration with a variety of IDEs.
- FindBugs - Open-source code analysis tool which is capable of detecting many different types of bugs in Java code.
- Yasca - An open-source code analysis tool which combines the functionality of many other open-source code analysis tools in order to provide quality and security metrics for code written in a wide range of languages.

The majority of existing code analysis tools are either commercial applications catering to companies, or open-source tools with limited functionality.

3. PROJECT PROPOSAL

We propose to design a process that will be used to quantitatively assess code quality. A user will upload examples of his or her code to a server, where our program will evaluate its internal quality by detecting code smells and its external quality through complexity analysis. We plan to focus on static, internal quality because we believe this approach allows for the quantification of source code's quality without becoming biased by different system architectures or environments. We plan to use code smells as an indicator of these metrics because they are specific and detectable, and because there is existing work which we can use as a starting point [12], [14]. Quality metrics to consider include readability, understandability, complexity, maintainability, and testability. Given the time constraints of this project, we intend to limit our scope to just two important metrics, understandability and maintainability, and include additional metrics if we meet our goals.

Understandability, a measure of how easy a code sample is to understand, can be estimated in part by the length of message chains, length of parameter lists, by keeping track of what fraction of variable names are dictionary words vs jumbles of letters and numbers, by analyzing the class structure of a program.

Maintainability, a measure of how easy a code sample will be to update and change, can be estimated in part by detecting and recognizing coupling between classes, duplicated code, and classes that are too large. Other indicative code smells include shotgun surgery, which describes the situation in which making one change requires lots of small changes to several classes, and divergent change, in which one class might need different changes for different reasons and indicates that two classes might be better than one.

These two components of quality are often underemphasized in classes, in which students work quickly and often haphazardly to meet project deadlines and never revisit their code after turning it in. However, understandability and maintainability are critical outside of the classroom, where many people need to understand code they did not write themselves, and be able to maintain it for years as the product evolves. Our tool would help students assess and improve these necessary software engineering skills, and help recruiters identify potential employees who will write maintainable, understandable code.

We plan to use crowdsourcing and machine learning to determine how much weight each code smell will contribute to the quality score based on the way a human would evaluate similar code samples. This will be done through a crowdsourced rating mechanism, in which users are given incentives to read through and rate training code samples. These ratings will then be compared with Our program will detect code smells and other issues, and machine learning will help us tune the parameters in our model to produce the overall grade.

Our tool will also provide a detailed score report. This report could then be shared with recruiters as a quick and objective evaluation of a candidate's programming ability. It could also be used as a tool for self-improvement, or by teachers as a grading tool. This report will also be summarized with a computed CodeScore, a score that reflects the overall quality of the code after analyzing both internal and external characteristics.

3.1 Anticipated Approach

Because of our familiarity with the language and conventions, we will focus our attentions on Java for the purpose of this project. We are comfortable with this choice because it is commonly used by companies for interviews, and it is taught at many institutions as a "beginner" language. We will also program the system in Java for several reasons: It will be easier to analyze Java code with the native language, Java can be run on almost any architecture, and the language lends itself to an elegant and easy to understand design.

Our overall system architecture is shown below. We will be leveraging Amazon Web Services (AWS) heavily for this project. One of the first things that we will set up is a code repository in the cloud on an EC2 machine. EC2 is a service that provides resizable computing capacity in the cloud. A user can upload their code to this endpoint, where it will trigger our application. To start, our application will analyze internal quality of a piece of code and generate reports for maintainability and understandability. Before our analysis engine runs, we will do a simple compilation of the code in the cloud to make sure that our input is well-formed.

To analyze the internal quality of a piece of code, we will have a worker class that will essentially act as a tokenizer of the source code and analyze it line by line. For example, after the source code is analyzed, the worker class could count the number of newline tokens to compute the "lines of code" metric. We will also check to see if variable names are apt for the situation. To this end, we will check to see if variable names are similar to English words. We can have a lexicon of the English dictionary that we can use to do this check. For each non-whitespace, non-keyword token that exists in the source code, we can do a check to see if that word is similar enough to a word in our lexicon. We will do this because variable names that have words in them are easier to understand than those variables without words in their names. We will also introduce a type of blacklist so that words that are not suited for variable names like "something" or "anywhere" will affect our metrics aversely. We will iterate over our tokens and count instances of if-statements, for loops, and while loops as well. We can also do simple analysis like how many parameters are in function signatures fairly easily. We will also look for daisy chaining of methods. To do this, we can go line by line in the source code and check to see how many method calls are being made at each line.

To watch over this, we will have a controller class that will coordinate a pool of worker threads. When a piece of software is uploaded to our endpoint, the controller class will spin off n workers, where n is the number of files in the piece of software that was just uploaded. Then each of the workers will compute their tasks as mentioned above. After all the work is done, the controller will then reduce all of the data computed by the workers and summarize the findings. This is our initial design. One drawback to this method is that it is largely centralized. If the server containing our controller code goes down at any point in time, we will lose all progress in our analysis. In the future, we can make this decentralized so that if the server goes down, not much work will be lost. Ideally, this process would be map reduce.

To analyze the external quality of the code, we will mainly do complexity analysis. This will be done by simply measuring the time it takes for certain snippets of code to execute. By tweaking the input parameters, our application will be able to roughly identify the theoretical algorithmic complexity of the code in question. An example of this is shown below in our evaluation section.

To then calculate basic maintainability and understandability scores, we will weight our metrics using weighted equations taking into account factors which contribute to both metrics. We will then combine them based on weightings specified by a user or on some default weighting.

Each feature examined to determine our metrics will be weighted by a parameter. These parameters will be tuned using machine learning techniques based on ratings for training code rated by volunteers in a crowdsourcing platform. We hope to do this via Amazon's Mechanical Turk. We will generate a large number of code snippets that each exhibit a small subset of code smells. We will then upload them to MTurk. There, each user will be prompted to look for these code smells and then give the snippet of code an overall score. We can check two things with this crowdsourcing:

- Which code smells are most often overlooked
- Which snippets of code have higher quality ratings (implying their code smells are not as bad)

We will then take these features from our user surveys and feed it through a machine learning algorithm via our learning module. Some learning techniques we are considering are linear regression, k-nearest neighbors, and kernel regression. We will then look at each code smell and see its relative information gain. From here, we can determine which code smells are more detrimental to software than others. This will also allow us to weight each of our calculated metrics in a more coherent way. Once we are able to weight our metrics, we will be able to compute the CodeScore.

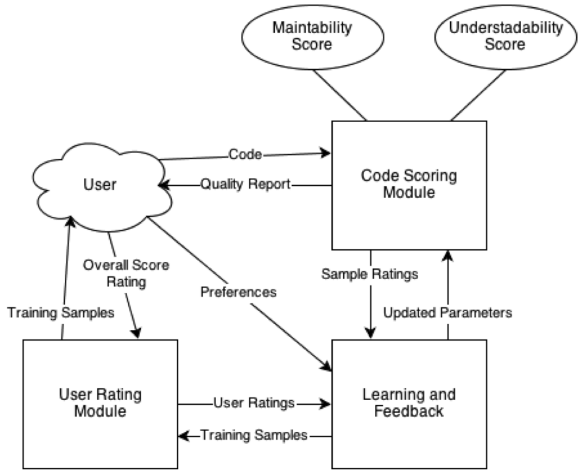

Figure 1: Block diagram of our system

3.2 Technical Challenges

The most significant technical challenge we will face is defining the rules for detecting each type of code smell. This is more a research problem than an implementation problem, and we will have to perform a series of experiments to fine-tune our detection algorithms. This will be especially difficult because in some cases, we will not have an existing detector to compare results with, but we can overcome this by comparing with human detection. We must then determine the best combination of feasible analyses that will yield an accurate and comprehensive reflection of internal code quality. This problem has been attempted many times, but the results leave much to be desired and are often limited to a single specific factor instead of providing a general assessment of talent. We hope to combine human and machine intelligence using crowdsourcing and machine learning to address this problem, but this strategy will present another challenge in terms of filtering through the results and incorporating them into our model efficiently and correctly.

Another technical challenge will be to determine the optimal weights for each of our code smells when computing the CodeScore. This will be difficult because we will have to analyze user surveys to see which code smells matter more. We will have to determine which machine learning approach will give us the most accurate weights for our metrics.

Additionally, we have to learn how to parse source code effectively. This is crucial to our project since we are going to need to analyze code not only at a character by character level, but also at a logical level. We need to come up with a way to calculate all of our metrics without having to jump around too much in the source code itself. Otherwise, our application would take a long time to execute.

Being able to analyze the code complexity of any arbitrary piece of software would be difficult as well. We would have to determine which inputs to send to the function in order to push it beyond its limits. After that, we would have to find a way to map the execution time to algorithmic complexity. This will have the tendency to use a lot of resources and time in our application, and we would ideally need to minimize this.

Depending on the way we conduct our crowdsourcing component, another technical challenge may be to standardize all the data that we get from this experiment. To feed this data through our machine learning algorithms, we will need to have discrete variables. This may not be possible if we let users answer questions in an open way. This may limit the kind of data that we can collect from users. Finding out how to achieve a good balance between standardized responses and open ended responses as well as analyzing them will be a challenge.

3.3 Evaluation Criteria

Since our project aims to quantitatively analyze code to determine how good it is, our evaluation criteria will revolve around checking how accurate our algorithm is in achieve in this goal. We aim to test:

- How accurately we detect code smells.
- How accurately we determine code complexity and scalability.
- How accurate our analysis is when compared to human analysis.

We hope that by measuring across these three areas, we will be able to validate that our application works as expected. As a goal for the project, we hope to hit 90% accuracy rates on the first two criteria and hope to beat human analysis at least 75% of the time.

For the detection of code smells, we can use precision (fraction of detected problems that are really problems) and recall (fraction of problems that are detected) to evaluate the performance. Our goal will be to achieve at least 90% for both precision and recall with each code smell detector. To do this test systematically, we will feed code through our application that has a fixed number of code smells. For example, we feed in a piece of software that has a total of 10 code smells. What we will then do is check to see how many code smells our algorithm detects in the code. Some things we will have to look out for will be false positives and false negatives. In both these cases, it will imply that our algorithm is not accurate as it should be. We hope that our application will have a $< 10\%$ rate for false results.

On a related note to code smells, we will also test the output of our machine learning algorithm to see if the various code smells that were found were weighted properly. To this end, we could compute the "code smell score" by hand and see if it matches the score that was computed by our application. We hope that these figures will always be within 5% of each other. This way, we can be certain that the output of our machine learning will be used correctly.

To test code complexity and scalability, we can employ similar tests and expect similar results. We can craft a small snippet of code that has a particular theoretical runtime. For example, we can input the following snippet of Java into our application:

```
public static void foo(int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      System.out.println(n);
    }
  }
}
```

This function has a theoretical runtime complexity of $O(n^2)$. Our application should be able to take this function and systematically determine that the complexity is $O(n^2)$. Since algorithmic complexities are largely discrete, our application can either determine the correct complexity or it cannot. Over a trial of 50 snippets of code, we hope to achieve a 90% accuracy rate. This means that we want at least 45 of our tests to match the theoretical algorithmic complexity. This will also give us a good indicator of how scalable the piece of software is.

The last variant of testing that we can do is human testing. Essentially, we can give small software packages, comprising of 10 source files each, to both our application and real developers. Our application will compute the CodeScore as well as all of the other metrics about the code. Real developers will then use their best judgement to analyze the code and assign each of the projects CodeScores based on how they found code smells, how they weighted those code smells, how they analyzed complexity, etc. Our goal is to get to a point where the discrepancy between the CodeScores generated by our application and the real developers happens only 25% of the time or less. To phrase it differently, we hope that at least 75% of the time, there will be a rough match between the computed CodeScores. (Rough match is determined by +/- 5%)

4. RESEARCH TIMELINE

Finally, we would like you to speculate about the pace of your research progress. This section need not be lengthy, we would just like you to specify some milestones so we can gauge your progress during our intermediate interviews. Let us follow through with our internal code quality analyzer example:

- ALREADY COMPLETED: Performed research on measuring internal code quality. Began looking into parsing techniques for source code and detecting code smells.
- PRIOR TO THANKSGIVING : Have a basic worker class and controller running. The controller should be able to spawn off workers when the server-side repo is updated.
- PRIOR TO CHRISTMAS : Have worker class implemented. Should be able to detect code smells and report back to controller. Controller should be able to reduce the output.
- BY THE START OF SPRING TERM : Have sample code snippets made for MTurk.
- BY THE END OF MARCH : Should have results from MTurk experiments. Implement some form of machine learning so that we can analyze the results.
- REACH GOALS : Investigate how to change design to map reduce. Look for more code metrics to measure and incorporate. Add additional code smells.
- COMPLETION TASKS : Verify implementation is meets requirements. Conduct testing mentioned in evaluation section. Complete write-up.

5. REFERENCES

[1] J Allen, S Collison, and R Luckey. Ohloh web site. https://www.ohloh.net/, 2009.

[2] Barry W Boehm, John R Brown, and Myron Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605. IEEE Computer Society Press, 1976.

[3] Barry W Boehm, Ray Madachy, Bert Steece, et al. *Software Cost Estimation with Cocomo II with Cdrom.* Prentice Hall PTR, 2000.

[4] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[5] Steve Christey. Plover: Preliminary list of vulnerability examples for researchers. In *NIST Workshop Defining the State of the Art of Software Security Tools*, 2005.

[6] The Web Application Security Consortium. Static code analysis tools. http://projects.webappsec.org/w/page/61622133/StaticCodeAnalysisList, 2012.

[7] Maurice H Halstead. *Elements of Software Science.* Elsevier Science Inc., 1977.

[8] ISO ISO. Iec 25010: 2011: Systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models. *International Organization for Standardization*, 2011.

[9] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[10] CVE MITRE. Common vulnerabilities and exposures. https://cve.mitre.org/, 2005.

[11] CWE MITRE. Common weakness enumeration. https://cwe.mitre.org/, 2006.

[12] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. http://www.archipel.uqam.ca/5165/1/Moha09-TSE.pdf, 2009.

[13] National Institute of Standards and Technology. Software assurance metrics and tool evaluation. http://samate.nist.gov/Main_Page.html, 2005.

[14] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. http://www.cs.wm.edu/~denys/pubs/ASE%2713-HIST-Camera.pdf.

[15] Robert E Park. Software size measurement: A framework for counting source statements. Technical report, DTIC Document, 1992.