Predicting Classes in Need of Refactoring: An Application of Static Metrics

Liming Zhao
Department of Computer Science
University of Kentucky
lzhao2@uky.edu

Jane Huffman Hayes
Department of Computer Science
University of Kentucky
hayes@cs.uky.edu

Abstract

This paper introduces a class-based approach to predicting refactoring candidates. Using a selected set of static metrics and a weighted ranking method, a tool was designed to predict a prioritized list of classes in need of refactoring. A study was designed and undertaken to compare the performance of the refactoring decision tool to that of human reviewers on the task of finding design problems that reduce class maintainability. The study results indicate that such a refactoring decision support tool can greatly assist the software team. The study also provided useful information on how to augment the functionalities of the refactoring tool.

1. Introduction

Software maintenance is costly. Research shows that software maintenance can consume 90% of the total cost of the software life cycle [Banker RD, 1993]. Refactoring is an approach to improving the design of software to make it easier to maintain without changing its external behavior. However, it is not practical for a software team to refactor a software application without considering the cost and deadlines of the project. In general, the following process is followed by a software team performing refactoring:

- Identify code segments that need refactoring,
- Evaluate or predict the possible cost and benefit of each refactoring,
- Develop a refactoring plan according to the results from the above steps and project resources, and
- Apply the refactorings.

We agree with Opdyke [1992] that the software team is the one who makes the final decision and performs the actual refactoring. However, proper tool support can make the process easier, faster, and more accurate. Commercial tools are available for applying refactoring automatically. However, effective tool support is still needed for the other three steps listed above.

The objective of refactoring is to reduce complexity of certain code units. Programmers refactor a code unit to either make it simpler or use indirections (such as extracting a method and calling the method) to hide the complexity. A code unit's complexity can increase due to its size or logic as well as its interactions with other code units. We focus on the previous case in this study, examining size and complexity metrics of code units.

Various techniques have been used to predict the maintainability of code and/or to identify code that is not easy to maintain (often called "bad smells"). These techniques include maintainability models, similarity measures, logic queries, etc. [Welker 1995; Simon 2001; Mens 2003].

Our maintainability (or refactoring target) prediction is class-based. Our assumption is that the code of a class is likely to have the same programmer(s). Moreover, code of a class is cohesive. If a programmer solves the maintainability problems of a class, the experience gained from that can directly benefit the programmer when he/she works on the next such problem.

The paper is organized as follows. Section 2 presents related work. Section 3 discusses our approach to predicting classes in need of refactoring. Sections 4 and 5 discuss the design and execution of the study, respectively. Conclusions and future work are presented in Section 6.

2. Related Work

Chidamber and Kemerer [1994] developed and evaluated several object-oriented (OO) metrics, referred to as the CK metrics, including WMC, CBO, and LCOM. Li and Henry [1993] hypothesized and validated the relationship between maintainability and a set of OO metrics, including five from the CK metrics and several of their own such as MPC (Message-passing coupling) and DAC (number of abstract data types defined in a class). Welker [1995] suggested measuring software's maintainability using a Maintainability Index (MI) which is a combination of multiple metrics, including Halstead metrics, McCabe's cyclomatic complexity, lines of code, and number of comments. Hayes et al. [1998] used textual complexity measures to locate segments of code that are difficult to change and thus need additional documentation. Their tool focused on the location of abnormal complexity. The tool also checked for structures such as forward declarations and recursion. Hayes and Zhao [2005] verified that Effort correlated with

maintainability and introduced and validated that the RDC ratio (the sum of requirement and design effort divided by code effort) is a good predictor for maintainability.

Fowler [1999] suggested using a set of bad smells such as long method to decide when and where to apply refactoring. Simon et al. [2001] measured the similarity between class members to decide which attributes and methods should be put together. They found that similarity/distance metrics can identify design abnormalities that disobey the cohesion principle and thus need refactoring. Mens, Tourwe, and Monuz [2003] designed a tool to detect places that need refactoring and decide which refactoring should be applied. They do so by detecting the existence of "bad smells" using logic queries.

Our approach differs from the above approaches in that it uses a cost-benefit analysis to prioritize the identified classes with bad smells That is, after identifying the locations/types of refactoring needed, we can predict the possible cost of the refactoring and its impact on code maintainability. We discuss our approach further in the next section.

3. Design of the Tool

The overall architecture of the maintainability decision support tool is shown in Figure 1. The tool is written in C and uses EDG's Java Front End[1]. There are four major components: Code Repository Analyzer, Maintainability Prediction Component, and Refactoring Planning Component. We discuss each in turn.

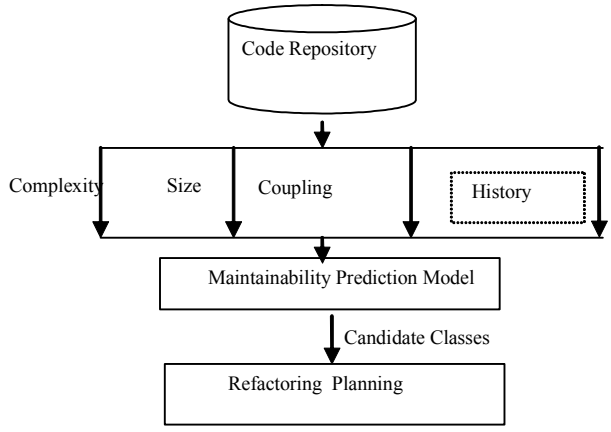

Figure 1. Maintainability decision support tool.

The Code Repository Analyzer is designed to parse the source code, discover structural characteristics, and collect metrics. We used EDG's JFE [EDG 2005] in building this component. JFE performs full syntax and semantic analysis of source code and represents the source code as a "high-level, tree structured" [EDG 2005] intermediate language – i.e., the Abstract Syntax Tree (AST) of the source code. The analyzer gets the necessary code information (complexity, size, and coupling measures) by traversing the tree.

The collected data are sent to the Maintainability Prediction Component. A weighted combination of multiple maintainability predictors (including a variant of MI (Maintainability Index)), complexity, and size are used to rank the classes. We do not combine the raw metrics. Instead, we use individual predictors to rank classes separately. A ranking matrix is generated for the classes to give a comprehensive view of the relative maintainability attributes of all the classes. At the same time, a weighted sum of the ranks of different predictors yields an integrated Relative Maintainability Vector (RMV). The RMV is used as the class-based refactoring priority list, to be given to the software team.

3.1 Metrics

We focus on size and complexity metrics at this point in time. Class interactions have been left for future work.

Halstead metrics [2] [Halstead 1977] are computed directly from operators and operands in the code. Halstead length (total number of operators and operands), vocabulary (total number of unique operators and operands), and effort are used to represent the length, understandability, and complexity of each class.

Weighted Methods per Class (WMC) is an important metric for OO. Assuming Class A has n methods, then:

$$WMC = \sum_{i=1 \text{ to } n} weight_i * method_i$$

where n is the number of all the methods in the class [Chidamber 1994].

We used two variants of WMC to represent the accumulated complexity and size of a class. Long Method Per Class (LMC) is used to count the number of large classes and Complex Method Per Class (CMC) is used to count the methods having an undesirable cyclomatic complexity [Pressman 2001]. The "undesirable" threshold can be set according to different project environments.

3.2 Class-based rank

Code of a class is designed to have high cohesion. To understand and/or modify one part of a class often requires one to read, modify, and/or test other parts as well. Further, if one member variable or one method is to

[1] We thank Dr. Stephen Adamczyk and EDG for their generous donation of JFE to our research program.

[2] We are aware that some researchers do not support the use of Halstead metrics. Our work to date has shown them to assist greatly in maintainability predictions.

be used in an implementation of new functionality, other members of the same class are highly likely to be used as well. Therefore, to save the overhead load and the cost of maintenance for new functionality, bad design problems or bad smells in a class should be fixed at the same time, if possible. In addition, classes that cost more to maintain should be improved first. Class-based rank (or relative maintainability) is used to help decide which class requires refactoring first.

When the software team tries to make a decision on which classes need refactoring most, the relative maintainability is more important to them than the actual metrics. This is because different metrics have different foci. Ranks of individual metrics give the software team an idea of how a class performs based on diverse attributes (size, complexity, etc.). Also, those classes that get a high rank from the most individual metrics should be a high priority for refactoring. Therefore, a comprehensive rank is needed -- we call it weighted maintainability rank (WMR):

$$WMR = \sum_{i=1 \text{ to } n} weight_i * rank_i$$

where n is the number of selected metrics.

In the study, we used the weight of 1 for every metric with good results. Varying the weight remains as future work.

4. Study Design

We were interested in seeing if our tool did a good job of predicting classes needing refactoring, in priority order. We designed a study to compare the tool to the predictions made by Java programmers. We wanted to study the following questions:

1. Can programmers identify refactoring candidates? Are their results similar to those of the tool?
2. What are the differences between programmer decision and tool decisions?
3. Do the programmers identify the same candidates for refactoring? Do they use common criteria? What kinds of problems are discovered?
4. How much time will programmers spend?
5. How should we improve our tool to help programmers/reviewers?

A set of Computer Science graduate students (some also work as programmers in industry part-time) were asked to review Java source code and identify classes needing refactoring. The code had been written as a graduate software engineering class project at the University of Kentucky. The project provides children with speech impediments an entertaining way to practice their speech therapy exercises. If the children vocalize in the proper pitch and loudness range, rewards are provided in the form of interesting animations

We first had the graduate student volunteers perform some pre-reading and complete a short questionnaire. Based on their answers, we decided if they could participate in the study (some students had no experience with Java or maintenance programming and were omitted). We then gave the selected volunteers (seven) instructions for the study. They were instructed to read the code and look for "bad smells" (defined in the pre-reading). We requested that they record any such bad smells found in a list. We allowed them to select a smell from a provided list or to fill in a problem they observed that was not in the list. The volunteers were asked to decide which class had the most serious problems and should be refactored first. They provided a report to us with a prioritized list of all the classes they thought needed refactoring. They also recorded the time that they spent reviewing/reading each class. Six students completed the study.

We also ran the tool to generate a class-based priority list (the tool does not generate an individual list of bad smells). The results from the manual inspection and the tool were then compared.

There are a number of threats to validity to our study. A threat to external validity is the use of students versus professional programmers. We tried to limit this threat by "vetting" the students. Also, we had a very small set of volunteers. A possible threat to internal validity is the use of various models to measure maintainability. We used the most widely accepted models possible, such as MI.

5. Study Results

All six reviewers turned in the two requested reports. The first one lists individual bad smells or design problems they identified in each class. The second one is a priority list of classes in need of refactoring. We observed that programmers/reviewers used different criteria to identify the most important bad smells. For example, some looked for unusual size, some looked for high complexity,

Table 1. Compare reviewers' selection with tool's selection.

Priority	Rv 1	Rv 2	Rv 3	Rv 4	Rv 5	Rv 6	To-ol
#1	I1	A	G	A	A	G	A
#2	F	W	T	P	E	H	LO
#3	W	LO	A	S	G		H
#4	G	V	L	W	H		P
#5	P	S		M	LO		F
#6					P		L

A - AnimationScreen, F-FileOutPut, L-LoginScreen, P- PitchAnalyzer, S-SessionReport, E- ErrorWindow, G-Graphic, LO - LoudnessAnalyzer, R-RegistrationScreen, T - TestScreen, F- FileInputHandler, H - HistoryReport, M - MenuScreen, W- Welcome, I1 – ImagePanel1, I2 – ImagePanel2, L – LoginScreen

Table 2. Statistics of collected metrics.

	Min	Max	Mean	Std. Dev.
Halstead_L	4	491	203.15	155.2
Halstead_V	4	36300	1894.7	8098.35
Halstead_E	0	74762	19353.3	23252.87
MI	3	125.33	62.63	26.99
LMC	0	11	3.75	2.59
CMC	0	5	0.65	1.27

Table 3. Metrics of Class AnimationScreen.

Halst-ead_L	Halst-ead_V	Halst-ead_E	MI	LMC	CMC
491	191	74762	37.49	5	2

and some were interested in dead code or duplicates. However, there was a significant degree of unanimity between reviewers, and between the reviewers and the tool on class selection.

Table 1 shows the class selection results of the reviewers. The class name abbreviations are shown above the table. The first column gives the priority rank. The first row represents six reviewers and the tool (Rv1 represents Reviewer 1, for example). Any of the other cells represent a class on which the reviewer made a decision. For example, the cell in the second row and the third column has the value "A" and indicates that reviewer 2 selected AnimationScreen as the class that needs to be refactored first (priority #1). Table 2 presents descriptive statistics for the metrics collected. We found that 50% (three out of six) of the reviewers and the tool agreed that class A is priority #1 for refactoring. In fact, 67% (four out of six) of the reviewers put AnimationScreen (class "A") on the priority list. We examined the metrics collected for AnimationScreen and found that it has the largest value of Halstead Length, Halstead Vocabulary, Halstead Effort, and the smallest value of MI (the smaller the MI, the worse the maintainability). In addition, LMC and CMC of this class were above average (Table 3). This indicates both the tool and the reviewers identified the most complex class.

Although only 5 or 6 classes were selected from a total of 20 by our reviewers, there are a significant number of classes that were selected by both a reviewer and the tool. For example, the classes AnimationScreen and LoudnessAnalyzer appear in both reviewer 2's and the tool's priority list (Table 1).

Further, we found that Reviewer 5's list had the most in common with that of the tool (Table 4). We did a Spearman rank correlation [Wessa 2006] on the common set of classes. The correlation value is 0.8, which is

Table 4. Comparison of reviewer 5's and tool's decisions.

Class	Reviewer 5	Tool Rank
A	1	1
H	4	3
LO	5	2
P	6	4

Table 5. Kendal tau Rank Correlation.

Kendall tau Rank Correlation	
Kendall tau	0.67
2-side p-value	0.308

significant. However, our sample size is small and we ignored data not in the common set.

We then did a Kendall tau Rank Correlation test [Wessa 2006] to measure the "overlap" of Reviewer 5 and the tool. The Kendall tau Rank gave a value of 0.67 and a 2-sided p-value of 0.308, which is not ideal [Table 5].

Reviewers spent significant time on understanding the scope of the source code, reading documents, and reading the source code in order to perform this study. Five reviewers reported their total time spent, with a minimum of 1 hour and a maximum of more than 3 hours.

The reviewers were asked to record the time they spent on each problem (bad smell) they found (only five did so). Table 6 indicates how hard it was for them to identify a design problem according to the amount of time spent. In the table, E stands for Easy (<1 minute spent), M for Moderate (1 to 5) minutes, and H for Hard (>5 minutes spent). E+M means one easy problem and one moderate problem were found. Each column represents a reviewer's ratings about problems he/she identified on classes. For example, the cell in the third row and the second column has an element E+M. This means that two design problems (one easy, one moderate) were identified by reviewer 1 in the class FileOutput. In total, 29 problems

Table 6. The difficulty level (approximate time spent) for reviewers in finding design problems in classes.

Class	Rv1	Rv2	Rv3	Rv4	Rv6
Animation Screen		E	M	E	
FileOutput	E+M				
Graphic	H		E		M
HistoryReport					M
ImagePanel1	E+M				
ImagePanel2	M				
LoginScreen		E	M		
MenuScreen				H	
PitchAnalyzer	E	E		E+M	
RigistrationScreen		M			
SessionReport		E		E	
TestScreen			E		
Welcome	E+M	E		H	
WriteToFile	M	E			

were recorded by the five reviewers (the sixth reviewer found 17 problems not shown here). Among them, only three (10%) were classified as Hard (more than 5 minutes to find). This supports the intuition that code reviewers/inspectors are more likely to find easy problems and it is possible for them to ignore those problems that are hard to find. Bad smells that are hard to find are still important, and we think that appropriate training and instruction may be necessary to help reviewers find these bad smells. This also provides evidence in favor of tool support, especially for industry sized projects. On the other hand, reviewers identified design problems such as dead code, duplicate code, and unused class that our tool is not yet able to identify. This encourages us to continue the design and development of the tool to that it can identify more individual bad smells.

6. Conclusions and Future Work

In the study, the results obtained from the tool (which focus on size and complexity) had a certain degree of agreement with those of the reviewers. This small study lends some initial support to the notion that complexity and size are among the major factors that add difficulty to programmers' comprehension of Legacy code and can be used to predict classes needing refactoring. We have shown that a priority list can be established in an automated fashion that contributes to cost-effective refactoring planning.

We also found that, although reviewers make some common decisions on which classes should be refactored first, they were often looking for different kinds of design problems. It was the case that 28% of the bad smells identified by the reviewers were not in the short list of bad smell types that we provided them. Code complexity appears to be the code characteristic that leads them to the same groups of classes. However, their own backgrounds and experiences led them to check for many different kinds of bad smells. This implies that relying on programmer review alone could lead to incomplete and/or inconsistent class identification. Therefore, automation can help improve the consistency, efficiency, and effectiveness of the code reviewing and refactoring decision process. Code reviewing is a time-consuming task requiring comprehension and even memorization of code. Appropriate tool support can help reduce this overhead cost.

In the future, we plan to predict the possible cost of the refactoring and its impact on code maintainability. Cost-benefit estimation prior to applying refactoring can assist with risk management, resource allocation, and planning.

7. Acknowledgments

Thanks to EDG Group for providing JFE. Thanks to the graduate student volunteers for participating in the study.

8. References

Chidamber, S., Kemerer, C., A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 1994. 20(6): p. 476–493.

EDG Group, JFE, a Front End for the Java Language ,http://www.edg.com/jfe.html, 2005

Fowler, M., Beck, K., Brant,J., Opdyke,W., Roberts,D., Refactoring: Improving the Design of Existing Code. 2001.

Halstead, M.H., Elements of Software Science. Operating, and Programming Systems Series, 1977.

Hayes, J.a.Z., L, Maintainability Prediction: A Regression Analysis of Measures of Evolving Systems. IEEE 21th International Conference on Software Maintenance, Budapest, Hungary, 2005.

Huffman, J., Burgess, C., Partially Automated In-Line Documentation (PAID). IEEE Conference on Software Maintenance, ICSM 1988,IEEE Computer Society: Phoenix, AZ, 1998: p. 60-65.

Hayes, J., Mohamed, N., Gao, T., The Observe-Mine-Adopt Model: An Agile Way to Enhance Software Maintainability. Journal of Software Maintenance and Evolution: Research and Practice, 2003. 15(5): p. 297 – 323.

Kafura, D., Reddy, R., The use of software complexity metrics in software maintenance. IEEE Transactions on Software Engineering, 1987. 13(3): p. 335–343.

Patel Veer, Building a Static Analysis Tool for Java Programs, Master's Project Report, University Of Kentucky, April 2004

Pressman, R.S., Software Engineering: A Practitioner's Approach. 5th ed. 2001.

Li, W., Henry, S., Object-oriented metrics that predict maintainability. Journal of Systems and Software, 1993. 23(2): p. 111-122.

Mens, T., Tourwé, T., Muñoz, F., Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring. Proceedings of the International Workshop on Principles of Software Evolution IWPSE 2003, 2003.

Opdyke, W.F., Refactoring Object-Oriented Frameworks, in Department of Computer Science. 1992, Univerisity of Illinois at Urbana-Champaign: Urbana,IL. p. 142.

Welker, K., Oman, P.W., Software maintainability metrics models in practice. Journal of Defense Software Engineering, 1995. 8(11): p. 19-23.

Wessa, P. (2006), Free Statistics Software, Office for Research Development and Education, version 1.1.18, URL http://www.wessa.net/