Contrasting Hotspots and Code Smells
in Context to Refactoring and Software Evolution

1. Introduction

Lehman's Law of Continuing Change (I) states that software will become progressively less satisfying to its users over time, unless it is continually adapted to meet new needs. This means that software must constantly evolve to meet new requirements. Additionally, Lehman's Law of Continuing Growth (VI) states that the functional content of software systems must be continually increased to maintain user satisfaction over their lifetime. User satisfaction is often defined by software flexibility and ease of use. These are often achieved through increased software complexity.

Both of these laws in the context of software evolution will often increase the code size, its complexity and require frequent code changes.

Refactoring is commonly applied to improve the software quality after a significant amount of features are added. Fowler defines refactoring as a change to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [1]. But, knowing where to apply the refactoring can be quite challenging. In practice, two techniques are used to start a refactoring effort: hotspots and code smells.

In this paper, I will explore the effectiveness of hotspot and code smell analysis tools in the context of software refactoring.

2. Hotspots

Hotspot detection is a quantitative technique based on analyzing patterns from a software change history records. Code hotspots represent large concentrations in the source code of specific measures such as, historical defects, frequent code changes or complexity. [2] These hotspots may indicate areas of the code that need refactoring.

In practice, detecting hotspots follows three basic steps.

i) Data Extraction Algorithm:
At this stage, the hotspot detection tool mines the history records to compile the data required for calculating the metrics.

ii) Calculation of Metrics:
At this stage, a quantitative analysis is performed using different measures like counting occurrences, correlations and probabilities. For example, calculating

correlation coefficients between change metrics and fault-prone files or determining the probability that a change will inject bug.

iii) Presentation of Results:
At this stage, the results are presented to the users either dynamically in the IDE or statically in the form of reports.

3. Code smells

The other technique used to identify parts of the code that could require refactoring is code smell detection. Fowler defines a code smell as a surface indication that usually corresponds to a deeper problem in the system. [3] In his book Refactoring: Improving the Design of Existing Code, Martin Fowler identifies twenty-two code smells. [1] Attach to a code smell are refactoring methods, that when applied can fix the smell. So unlike hotspots which only outlines potential areas that may need refactoring, code smells generally have one or more refactoring method associated to it.

4. How are code hot spots and code smells used in refactoring

Software refactoring, in the context of software evaluation and of the Lehman's Laws discussed in the introduction, is often done to reduce code complexity after multiple changes are made and new functionalities are introduced. Therefore, refactoring is a time consuming process and when not planned properly it can be an expensive venture that can derail deliverables. A software refactoring effort generally includes the following steps:

i. Code smell and/or hotspot analysis
ii. Identify which part of the code needs refactoring using results from steps i.
iii. Select refactoring techniques
iv. Apply refactoring techniques
v. Apply regression testing to the refactored code

Given the many steps and their complexity, it is important to plan extensively and identify the parts of the code that need to be refactored correctly. The first two steps serve precisely that purpose. The rest of the paper focuses on these two steps, and on the software that helps with their analysis.

5. Refactoring tools and empirical studies of their effectiveness

It is not feasible for large software to do the above steps manually. That's why programmers rely on tools.

5.1 Code smell analysis tools

Code smell tools can be divided into three categories: fully-automatic, semi-automatic and fully-manual. In the fully automatic version, the code smell detection and refactoring is done without user interaction. But this comes at a cost; as identified by Erica Mealy in her paper *Evaluating Software Refactoring Support Tool*, "The lack of user input into the process causes the introduction of meaningless identification names, lack of customizability and negative impacts on a user's current understanding of the system."[4] Semi-automated refactoring tools attempt to address some of the problems encountered by fully-automated tools by allowing the user to be part of the refactoring by inputting data (for example variable and method names) in the process. Under the fully-manual category, the software only detects the code smells and lets the user make all the refactoring decisions. In her study, Erica Mealy evaluated six refactoring tools: Eclipse, IDEA, JBuilder, Condenser, jCOSMO and RefactorIT. In her quantitative data summary, she summarized her finding using height categories with the Refactoring Specific Category carrying the most weight. That category evaluates each of the refactoring stages (detection, proposal and transformation) and the complexity of the refactoring supported. The average score for that category was 5.85 out of 15, which she observed is low, and is mainly explained by the fact that these tools do a fairly good job at single-stage refactoring and are less capable of performing multi-stage refactoring or complex refactoring. Which she says illustrates that none of the tools fully support the refactoring process as a whole.

In Tom Mens and Tom Tourwe paper titled *A Survey of Software Refactoring,* [5] come to a similar conclusion. By first noting that, "the nature of object-oriented principles make some seemingly straightforward restructuring surprisingly hard to implement. The encountered difficulties typically have to do with the inheritance mechanism and, more in particular the notions of dynamic binding, interfaces, subtyping, overriding, and polymorphism." They end their paper by recognizing the need for the current tools to be more generic, scalable and flexible.

Zhenchang Xing and Eleni Stroulia in section 6.3 of their paper *Refactoring: How it is and How it Should be Supported – An Eclipse Case Study,* outlined a similar problem. [6] By noting that code smell tools are not yet capable of supporting high-level refactoring: "Modern IDEs, such as Eclipse, support the most commonly used, low-level refactoring […]. But they do not support downcast type and information hiding refactoring."

After experimenting with Eclipse, IntelliJ JDeodorant and RefactorITby Francesca Arcelli et al. it identified another shortcoming of refactoring tools, which is "the possibility to introduce defects or any other kinds of problem in the code after refactoring." [7]

As demonstrated by these four papers, many of the refactoring tools relying on code smells are capable of identifying code smells and performing simple (or single-stage) refactoring. But fail at performing multi-stage refactoring. Therefore, programmers should be cautious when using fully-automatic tools since they may introduce unwanted or meaningless logic into the code. A better solution could be to run these tools in manual mode in order to detect the code smells. Then let the programmer confirm the validity of the smell and perform the more complicated refactoring manually. This seems especially valid for large and complex software.

5.2 Hotspot analysis tools

An alternative to using code smells, is to use hotspot analysis. This method relies on analyzing historical changes in the code base. But unlike a lot of the code smell tools, it doesn't suggest how to refactor the code.

There are very few research papers written on hotspot tools, which I speculate is because the better tools are developed internally based on the coding and bug reporting practices of each companies. Also, in order to effectively study hotspot tools academia needs to access the large code bases and historical change files. Technology companies are most probably very reluctant to share that sensitive information.

I found only one relevant paper *Code Hot Spot: A Tool for Extraction and Analysis of Code Change History* [8] written by Emerson Murphy-Hill. In *section E. Published Tools,* the author notes that "there have only been a few tools published that extract and analyze change history data." He goes on to describe four hotspot tools: HATARI, ROSE, APFEL and EMERALD. He concludes his paper by identifying three advantages for developers of using hotspot tools:

a) Make better decisions about handling change prone software files.
b) Use the information from the hotspot analysis to prioritize verification activities. In order to allocate greater efforts to risky files.
c) Target efforts at code improvement such as refactoring.

In an article written by two engineers at Google, they explain their motivation for creating the tool, how it was built and how it is used. [9]

Their main motivation was to inform their developers that they were working on a section of the code that has been historically problematic. Given the size of their code, it wasn't reasonable to assume that the developers could remember which part of the code created issues repeatedly. Second, as their code base and teams increases in size, it serves as a communication tool between the submitter and reviewer that extra attention should be given to review process. As we can see, their motivations were very similar to the advantages identified above by Emerson Murphy-Hill in his paper.

In order to develop their tool, they first looked at different approaches. The first was bug prediction using machine-learning and statistical analysis. But they found that it generally led to flagging innocuous complex code in addition to code hotspots. After reviewing the outstanding research on predicting hotspots from historical change files they decided to settle on that approach. In the first version, they implemented Rahman's algorithm [10], which works by, "ranking files by the number of times they've been changed with a bug-fixing commit." Although that approach worked very well, it tagged files as hotspots that were recently re-worked and bug free. To address that issue, they rolled out a second version that attached a time stamp to the historical files that served to identify how old each bug-fixing commit was. As time grew, its influence went towards 0, so it was removed from the hotspot list.

Google implemented their tool in their review system, so the files that were tagged as a hotspot would display a warning when the reviewer logged into the review code. Which they hoped would encourage them to spend more time reviewing or seeking the help of a more senior developer to assist in the review.

Although there is limited research on hotspot tools, the two articles above indicated that using hotspot detection offers a very effective way to identify problematic areas of the code. These are generally the complex parts of the code base and offer a good starting point for a refactoring project.

6. Conclusion

The empirical studies presented in this paper demonstrated that code smell tools seem to work best for simple refactoring with little ripple effect throughout the code. But for the more complex (or multi-stage refactoring) they are either not supported or not supported very well. This is because for complex code bases or large systems, refactoring cannot be applied mechanically, and design also needs to be considered.

The research papers also showed that a better approach is to use a hotspot detection tool in order to identify parts of the code that need to be refactored. Since the more complex part of the code tend to exhibit the most hotspots. Therefore, the tool will naturally outline areas of the code that need to be refactored. But using hotspot tools can be challenging since very few are available and thoroughly reviewed by the research community.

A practical approach could be to use code smells and hotpots in combination. So use code smell tools to detect and fix simple refactoring (single stage refactoring with low ripple effect). Then, use hotspot tools to detect complex or flawed code, which generally tends to appear frequently in the change record files. Once they are identified they can be analyzed and refactored manually.

References

[1] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Longman Publishing Co.

[2], [8] Emerson Murphy-Hill, *Code HotSpot: A Tool for Extraction and Analysis of Code Change History*, North Carolina State University.

[3] http://martinfowler.com/bliki/CodeSmell.html

[4] Erica Mealy and Paul Strooper, *Evaluating software refactoring tool support*, The University of Queensland.

[5] Tom Mens, Member, IEEE, and Tom Tourwe, *A Survey of Software Refactoring*, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 30, NO. 2, FEBRUARY 2004.

[6] Zhenchang Xing and Eleni Stroulia, *Refactoring Practice: How it is and How it Should be Supported – An Eclipse Case Study*, University of Alberta.

[7] Francesca Arcelli Fontana et al., *On Experimenting Refactoring Tools to Remove Code Smells*, University of Milano-Bicocca.

[8] Chris Lewis and Rong Ou, *Bug Prediction at Google*, http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html

[9]https://www.researchgate.net/profile/Earl_Barr/publication/221560306_BugCache_for_inspections_hit_or_miss/links/0912f509261c53c9c4000000.pdf