Summer 6-2020

Increasing the Trust In Refactoring Through Visualization

Alex Bogart
Rochester Institute of Technology

Eman Abdullah AlOmar
Rochester Institute of Technology

Mohamed Wiem Mkaouer
Rochester Institute of Technology

Ali Ouni
University of Quebec at Montreal

Recommended Citation

Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2020. Increasing the Trust In Refactoring Through Visualization. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20). Association for Computing Machinery, New York, NY, USA, 334–341. https://doi.org/10.1145/3387940.3392190

Increasing the Trust In Refactoring Through Visualization

Alex Bogart
alex.bogart@mail.rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Eman Abdullah AlOmar
eman.alomar@mail.rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Mohamed Wiem Mkaouer
mwmvse@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Ali Ouni
ali.ouni@etsmtl.ca
ETS Montreal, University of Quebec
Montreal, Quebec, Canada

ABSTRACT

In software development, maintaining good design is essential. The process of refactoring enables developers to improve this design during development without altering the program's existing behavior. However, this process can be time-consuming, introduce semantic errors, and be difficult for developers inexperienced with refactoring or unfamiliar with a given code base. Automated refactoring tools can help not only by applying these changes, but by identifying opportunities for refactoring. Yet, developers have not been quick to adopt these tools due to a lack of trust between the developer and the tool. We propose an approach in the form of a visualization to aid developers in understanding these suggested operations and increasing familiarity with automated refactoring tools. We also provide a manual validation of this approach and identify options to continue experimentation.

CCS CONCEPTS

• **Software and its engineering → Software organization and properties**.

KEYWORDS

Software maintenance and evolution, Refactoring, Visualization.

ACM Reference Format:
Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2021. Increasing the Trust In Refactoring Through Visualization. In *Proceedings of ACM Conference* (*Conference'17*). ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Quality is the key of longevity for any software, mainly dependent upon architecture, complexity, and a myriad of other nonfunctional attributes. Given the inevitability that, along several cycles and releases, the growth of software, in terms of size and services, will cause its initial, well-engineered design decay. Refactoring has been defined as the *de facto* of preserving the software's design by reverse the negative effects of continuous development without altering the software external behavior.

Refactoring refers to improving the quality of a code base after design has concluded and development is underway by altering the way in which the program is structured without altering external behavior [10]. It optimizes the high-level perspective of the system by applying design-based improvements (such as relocating and restructuring fields, methods, and classes) and by enhancing the code's readability and modifiability. The driving force behind these changes is either improving static metrics and quality attributes, including coupling, cohesion, and complexity [4, 15, 17–19] or by removing code smells, such as shotgun surgeries, god classes and blobs [7, 9, 10, 16, 22, 23]. Many modern integrated development environments (IDEs) have commonly deployed refactoring operations like Rename and Move Method refactoring as built-in functions, eliminating the need to handle any side-effects of manual refactoring that might affect the system's behavior. In other words, this functionality uses pre- and post-application checks to verify that no semantic changes were introduced.

Refactoring can be manual or automated. As software grow in size and complexity, manual refactoring becomes complex as well, subjective, and even error prone. Therefore, several studies have been exploring the automation of refactoring and several tools and frameworks have been proposed as to support developers in automatically recommending refactoring operations [4, 7, 11]. Particularly, JDeodorant [7] stands as one of the popular tools that have been provided as an Eclipse plugin for the community. It automatically detects design anti-patterns, and offers a wide variety of possible refactorings to correct them. Developers, are then responsible of choosing the most adequate refactoring operations according to their design choices and preferences.

The promise of automated approaches is to greatly increase the efficiency and accuracy of the refactoring. However, refactoring tools see considerably less than expected use among modern developers [2, 21]. Amidst various complaints regarding these tools, many developers simply do not trust them, either in their assessments, behavior preservation, or in the many changes they introduce to the existing design, because,

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

although developers are eager to optimize their code base, they still want to recognize their existing design. Thus, existing researchers seem to be fighting an uphill battle as they increase the accuracy, and so the automation of refactoring, while simultaneously trying to convince developers to deploy them.

As our aim, in this paper, is the support of developers to becomes more familiar with automated refactoring. This would not have been possible without the original developers of JDeodorant making their tool open-source as well, and for that we are grateful. In addition to this extension itself, we present the design process that led to its creation as a contribution. The development of the extension itself serves as the precursor to the validation, essentially serving as a proof of concept.

Specifically, this extension is built upon the popular refactoring tool JDeodorant [7], extending its interactivity as follows: we define a fine-grained visualization of multiple refactoring operations suggested to the user by the primary refactoring tool, enabling developers to better understand the impact of applying these refactorings, as well as conflicts that might arise during their application. This introduces the possibility of approximating simultaneous execution of multiple refactoring operations, rather than serially executing refactorings without a complete plan, helping to reduce the human effort required to optimize the system.

For the sake of simplicity, in this work, the term "extension" shall refer to the functionality we developed and added to JDeodorant, while "tool" shall refer to a fully-fledged refactoring tool, such as JDeodorant.

To aid in this endeavor, this work provides the following contributions:

- Firstly, an extension allowing the simultaneous execution and the visualization of multiple refactorings.
- Secondly, an analysis of this extension's effect in reducing the users' and performance effort when refactoring existing software systems, in comparison with the original tool.

The paper is structured as follows: Section 2 discusses the advantages and limitations of existing refactoring tools. Section 3 details the different features of our extension. Section 4 outlines the research questions, used to drive our validation, and discusses the results. Section 5 captures any threats to our study validity, before concluding with Section 6.

2 RELATED WORK

Detection tools can be defined by three levels of functionality. At the first and lowest level, most detection tools work by detecting code smells. By analyzing mostly static metrics, a tool can identify classes that would benefit from being refactored with varying degrees of accuracy [27]. Tools that solely identify metrics without analyzing them cannot really be considered refactoring tools in and of themselves. A study by Fernandes et al. [6] of several tools' recall and precision percentages showed that, compared to a human-defined code smell reference list, certain tools identified less than ten percent of a given code smell, while others earned perfect 100% identification scores. It should be noted that this does not directly compare with regards to human accuracy in code smell detection, as there is often little agreement among developers as to what constitutes a significant code smell [6]. On the same note, Marinescu's metric-based detection strategy had a recall of 100% and a precision of 71% compared to God Classes identified by human subjects [14].

The second level of functionality is the candidate suggestion. These tools will offer suggestions on how to refactor the identified code smells. This can be particularly useful to developers without a comprehensive understanding of the code base, as it can offer solutions the developer may not have considered, such as creating an interface for similar classes or identifying classes with high-efference couplings that would make ideal Move Method targets [12].

The third level of functionality is the candidate ranking. These tools will take their candidates and, utilizing predictive algorithms, attempt to establish a confidence rating for which candidate would be the ideal implementation for the given project. Several approaches for this have been presented, such as analyzing previous changes or system complexity [26]. Several approaches were implemented as JDeodorant, a refactoring tool that combines detection of a set of code smells and suggests corrections to the developer. JDeodorant Eclipse plug in will be detailed in the following section.

JDeodorant is a well-known refactoring tool in the form of an Eclipse plug-in [25]. It can identify a number of code smells in compilable Java programs, including God Class, Feature Envy, Long Method, Duplicated Code, and Type Checking (also known as Switch Statement) [9]. In addition, JDeodorant automatically generates potential refactoring operations for these code smells, ranking them (when necessary) by overall impact on the code base. Certain refactorings can be visualized in a UML-style diagram. The tool ties into Eclipse's refactoring functionality, as well as providing its own, allowing the user to preview and implement a selected operation automatically. It also comes with a package view visualization, allowing the user to identify which classes in the program have the highest concentration of a particular type of code smell.

Other refactoring tools, such as inFusion and iPlasma, have different approaches to detect code smells. They utilize metric-based formulas to identify code smells [9]. In slight contrast, JDeodorant identifies God Classes by utilizing clustering to identify groupings of code entities that would improve overall system design, which is unrelated to the overall size of the class [8]. While God Classes do tend to be larger in comparison to others, this is not inherently an indicator of a God Class. This clustering approach differs from the purely metric-based approach of inFusion, which utilizes class cohesion and complexity to identify God Classes [9]. For Feature Envy, the number of calls within function executions is measured as distance, and refactorings are recommended that reduce the overall "distance" within the program [7]. Its suggested refactoring opportunities are based on methods whose efference would decrease if moved to another class [9].

Recently, the raise of refactoring detection tools [], has allowed the mining of how developers actually refactor their code. Empirical studies, analyzing the commit messages associated with refactorings, have shown that refactoring can improve quality attributes such as complexity, size, cohesion, coupling, maintainability, and reusability [1, 3, 20, 24].

3 EXTENSION FEATURES

As we have established, trust is the primary concern of this work, our proposed solution comes in the form of an experimental visualization whose ultimate purpose is to increase the level of trust between the developer and the refactoring tool. The majority of the technical design decisions were made to either keep with convention or increase ease of development. Java was chosen as the primary development language due to its purely object-oriented nature and refactoring techniques' predilections towards object-oriented languages. Fernandes et al. [6] found that out of 84 code smell detection tools, Java dominated, both in terms of languages that the tools analyze as well as languages the tools were developed. They also found that an equal number of the tools were plug-ins and standalone programs. Rather than a standalone application, we decided to build off an existing tool, eventually deciding on the Eclipse plug-in, JDeodorant [25]. JDeodorant identified well-known code smells in Java projects and then recommended the appropriate refactorings. For example, in order to eliminate feature envy code smells, JDeodorant resolved it by applying move method refactorings. JDeodorant proved to be an ideal candidate as there are few completely open-source refactoring tools as popular as JDeodorant. Additionally, the Eclipse plug-in library is fairly substantial, and would provide an existing base for development, limiting potential issues related to building a new application from the ground up. In this work, our extension only supports two refactoring operations, namely, Extract Class refactoring and Move Method refactoring, and restricted to God Class and Feature Envy code smells.

The extension to the JDeodorant plug-in can be accessed by selecting any code smell to refactor from the "Bad Smells" dropdown. After the project selection has been parsed and code smells have been identified, a new column labeled "Implement?" will be visible on the far right. Checking one of these boxes will add the selected operation to the visualization, as seen in Figures 1 and 2. For God Classes, selecting a parent row will also select all its children. These operations often include overlapping elements, so this is most likely useful for scouting potential refactoring combinations. The selection can be cleared using the "Clear Selected Candidates" button. Note that this is the only way to remove operations from the visualization; selecting a new project will not reset the selection, as some users may want to visualize refactorings across multiple projects. It will also only clear God Class refactorings or Feature Envy refactorings, depending on which view is selected at the time.

3.1 Selection of Multiple Refactorings

If we want developers to better understand the impact of refactoring in their code, it is important to add a *preview* feature, which allows the vizualization of what would be the refactored design of the code, once many refactorings are applied. Developers should be able to view a UML-styled view of the results of a single or multiple refactoring operations, that they select prior to their execution. Utilizing the embedded code in JDeodorant to generate diagrams, we experimented with different methods of representing this information, as well as how prominent and detailed each piece of information could be. As the multiple execution, we extended the user interface to allow developers to select multiple refactorings, instead of only one at the time. Then we updated the refactoring engine to schedule their execution. Since the tool's recommended refactorings are independent, then the order in which they are scheduled does not matter.

Our main motivation behind visualizing the impact of multiple refactorings, and allowing their simultaneous execution, is that, developers rarely apply only a single refactoring operation to large, enterprise-level systems. Even an experienced user might have difficulty planning such a task. For a developer unfamiliar with a given program, trying to comprehend what dozens of operations would do to the program would be a daunting task, and would more than likely prompt them to either spend significant time analyzing and understanding the system, or blindly trust the assessment of an algorithm. This new visualization could display all these refactorings at once, simplifying their details into core concepts such as extraction, relocation, and merging, allowing the user to deep dive into a certain operation when necessary.

3.2 Visualization of Multiple Refactorings

A perhaps overlooked issue with refactoring tools is their inability to be used in conjunction with one another. By comparing the refactoring candidates generated by multiple tools, identifying the conflicts, and visualizing the combination of the valid refactorings, developers could utilize multiple tools to aid in their refactoring processes, rather than a single one. This would allow for more coverage and suggestions, and could potentially increase the user's confidence in the candidates if multiple tools' suggestions were the same.

The refactorings used in the visualization are selected from those generated by JDeodorant. From an internal standpoint, JDeodorant's refactorings could be tested for conflicts. However, with this feature, any other tool capable of exporting its refactoring candidates could be tested. Additionally, it allows for testing across multiple, separate refactoring tools.

3.3 Refactoring Conflict Visualization

The final design is centered around combining multiple refactorings into a single, presentable view. To visualize the selected refactorings, click the "Visualize Selected Candidates" button. This will open the Code Smell Visualization view, as seen in Figure 3. The entities in this particular diagram are classes, with the operations represented as arrows between

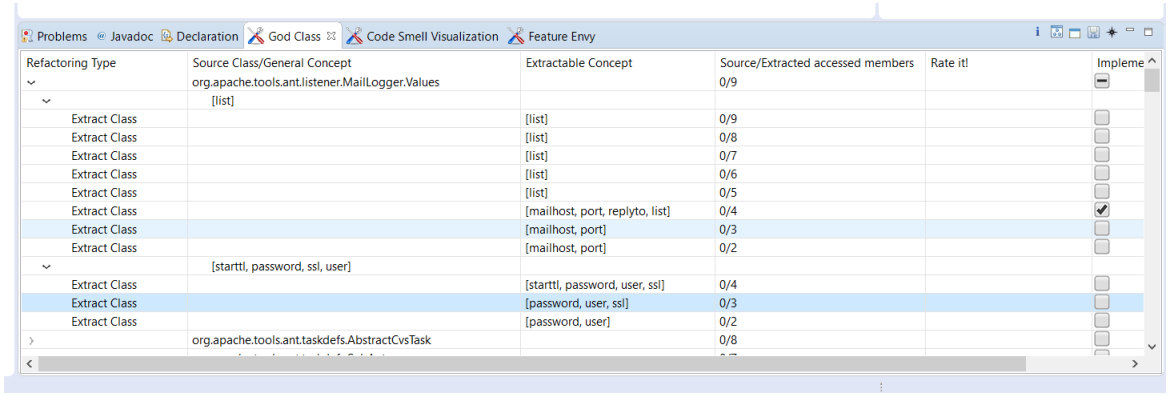

Figure 1: Selecting an Extract Class refactoring to visualize in the extension to JDeodorant

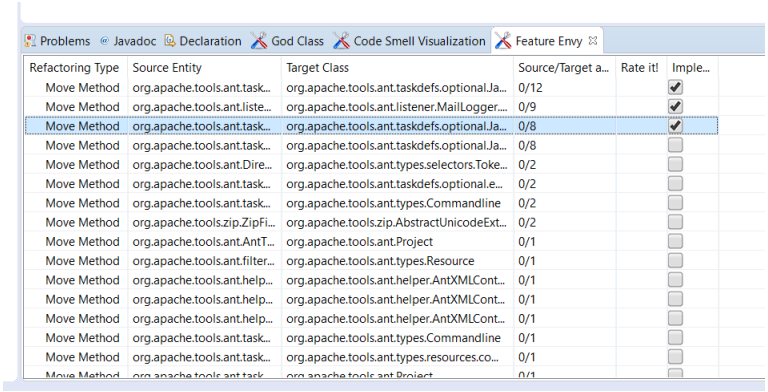

Figure 2: Selecting multiple Move Method refactorings to visualize in the extension to JDeodorant

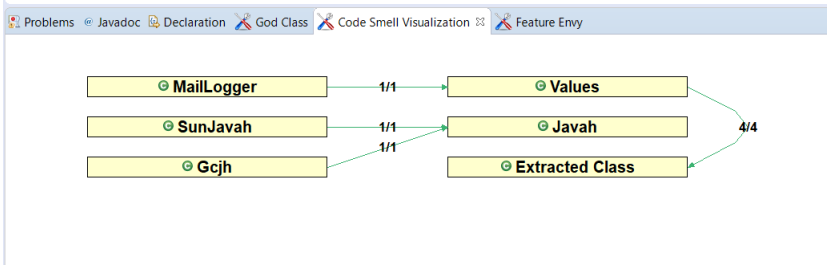

Figure 3: Visualization of the refactoring operations selected in Figures 1 and 2

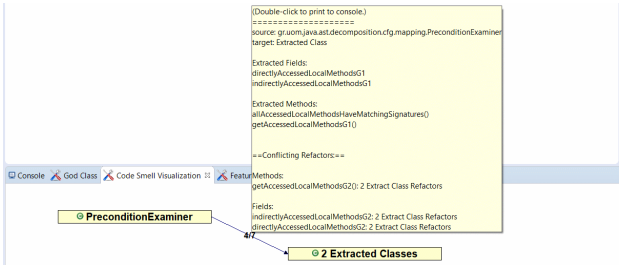

Figure 4: Visualization of conflicting Extract Class refactorings, with a tooltip

the classes. The scope of the diagram is the directly affected classes. The number of entities (methods and fields) modified is represented on the connecting line, with the number of non-conflicting entities over the total. If a class has Extract Class refactoring performed on it multiple times, the extracted classes will be visualized in a single location for simplicity's sake. The connections will change color based on the number of conflicting refactorings. If there are no conflicts, the arrow will be green. Black indicates some conflicts, while red indicates over 75% of the entities involved in that class' refactorings conflict with other operations. Hovering over either the source class or the connection will cause a tooltip to appear displaying the details about the entities involved in the operations. It also displays which types of operations the conflicting entities are a part of. An example of a visualization with conflicts can be seen in Figure 4. This information can be copied for later use. The class diagrams are simple rectangles, and are arranged in two columns to minimize the chance that connections will overlap with another element of the diagram, though this is far from ideal.

4 EXPERIMENTS

In order to evaluate our extension, we performed a set of experiments using 8 open-source systems. In this section, we start with outlining our research questions, describe how we address them, and then we discuss the obtained findings. We have identified two research thrusts that address the applicability, the performance in comparison to existing refactoring approaches, and the usefulness of the extension. Our research questions are as follows:

RQ1: To what extent can our approach help the simultaneous selection and execution of multiple refactorings to developers?

The specific benefits of refactoring tools are difficult to quantify due to the unique nature with which developers refactor and utilize refactoring tools. While these can be inferred and elicited through surveys and human studies, it is inherently impossible to completely understand the inner workings of every developer, divided into groups by age, experience, and personal preferences. Therefore, if causation is out of reach, one can at the very least identify correlations. Our goal is not to prove that this approach has a statistically proven benefit for developers. Rather, we intend to discover through concentrated evaluations and individual, written responses if there is a correlation between the use of this type of visualization and noticeable benefits to refactoring, including both time spent refactoring and the developer's willingness to use the given refactoring procedure. This serves as the first step in showing that this avenue of research may yet bear fruit.

RQ2: Can the use of this extension make the suggested refactorings more trustworthy in the eyes of the developer?

As the ultimate goal is to increase the level of trust between the developer and the tool, we asked a select number of developers what their impressions of the extension and its visualization were, and whether or not it helped them to better understand the refactorings proposed by JDeodorant. This particular wording was chosen due to "trustworthiness" being a difficult concept to quantify. Additionally, this provides the user with the option to directly compare two states (with and without the extension).

Table 1: Selected Projects

Project	Release	# Classes	KLOC	# CodeSmells
Xerces-J	v2.7.0	991	240	91
JHotDraw	v6.1	585	21	25
JFreeChart	v1.0.9	521	170	72
GanttProject	v1.10.2	245	41	49
Apache Ant	v1.8.2	1191	255	112
Rhino	v1.7R1	305	42	69
Log4J	v1.2.1	189	31	64
Nutch	v1.1	207	39	72

4.1 Projects Under Study

To evaluate the extension, we used a set of well-known, open-source Java projects. We applied the extension to eight of these projects: Xerces-J, JHotDraw, JFreeChart, GanttProject, Apache Ant, Rhino, Log4J, and Nutch. We selected these eight systems for the evaluation because they range from medium to large in size, they are open-source, they have been actively developed over the past ten years, and their development has not experienced slowdown due to their design. We used multiple projects rather than a single one to mitigate the issue of a project being easier or harder to refactor than others. Table 1 provides descriptive statistics about these eight programs.

4.2 Qualitative Analysis

For the eight Java projects chosen, a combined total of ten classes with potential code smells were manually identified in each project. The code smells were restricted to God Class and Feature Envy, as these were the only code smells supported by the extension at the time of writing. These projects were then grouped into pairs.

Eight developers experienced with refactoring operations participated in this evaluation. These were all developers with experience using refactoring tools, as we desired the input of those familiar with this domain rather than those with no experience, as the extension had not been developed with that use case as its primary goal. All that was made known to this author was the number of participants, so a folder was constructed for each participant, containing a set of written instructions, a video tutorial demonstrating the extension's use, and the following requisite files. Each developer was provided with a copy of JDeodorant with the extension and two pairs of projects (four of the eight projects in total). They were each given a list of the classes containing the code smells, and asked to refactor the projects. The developers were instructed to refactor one pair of projects using the tool and visualization (extension), while the other pair were to be refactored without the aid of the visualization (see Table 2). The developers were asked to record how long it took to refactor each pair of projects.

Table 2: Distribution of Programs Among Developers

Dev.	Programs	Refactoring Tool
Dev-1	JFreeChart, JHotDraw	JDeodorant
	Apache Ant, GanttProject	JDeodorant + Ext
Dev-2	Apache Ant, GanttProject	JDeodorant
	JFreeChart, JHotDraw	JDeodorant + Ext
Dev-3	Xerces-J, Rhino	JDeodorant
	Log4J, Nutch	JDeodorant + Ext
Dev-4	Log4J, Nutch	JDeodorant
	Xerces-J, Rhino	JDeodorant + Ext
Dev-5	JFreeChart, JHotDraw	JDeodorant
	Apache Ant, GanttProject	JDeodorant + Ext
Dev-6	Apache Ant, GanttProject	JDeodorant
	JFreeChart, JHotDraw	JDeodorant + Ext
Dev-7	Xerces-J, Rhino	JDeodorant
	Log4J, Nutch	JDeodorant + Ext
Dev-8	Log4J, Nutch	JDeodorant
	Xerces-J, Rhino	JDeodorant + Ext

There were no requirements given on which projects to refactor first.

Additionally, each developer was provided with a short questionnaire. This asked for the developers to rate the features of the extension, provide feedback on its features and potential improvements and additions, and to describe how this particular extension affected both their refactoring process and opinion of trust in the refactoring tool.

4.3 Refactoring Times

The refactoring experiment showed little correlation in refactoring times between developers asked to refactor the same projects with the same tools. Differences between refactoring times tended to vary between ten and 15 minutes, with extremes as low as five and as high as 29.

As shown in Figure 5, the total time to refactor the projects decreased when using the visualization in all but one instance. Feedback from the developers indicated universal appreciation for the visualization. Responses indicated that it helped principally with planning which refactorings to implement, as well as understanding what the changes to the system design would be. The ability to plan multiple refactorings at once, or "batch fix," seemed to help significantly with refactoring times, even when the developers needed to perform additional refactoring afterwards. One developer noted that being able to select multiple candidates alone was a benefit. While some claimed that using the tool was easier than their static analysis methods, others acknowledged its benefits while still retaining their preference for manual refactoring, at least so far as defining refactoring operations.

4.4 Questionnaire Results

To answer our research questions, we analyzed the responses from the participating developers. The developers noted that the visualization was able to identify and prevent several conflicts. However, on at least two occasions an operation caused the code to break, forcing the developer to refactor that portion manually. Expanding the functionality of the conflict detection with additional checks may help to prevent these issues.

A number of developers commented on the lack of information displayed by the visualization. In some cases the defects were not adequately described. This makes sense, as the extension only visualizes the solutions to be implemented, not the inherent problems with the classes. This made experimenting with the selected visualizations to find the right batch somewhat more time consuming and difficult than was necessary. One developer noted that this might also be mitigated prior to visualization if the extension was able to easily convey the details of the suggested refactoring candidates. This is possible at the moment with the "visualize code smell" feature (located by right-clicking on a refactoring candidate), but this process is also slow and cumbersome when dealing with dozens of potential refactorings for a single class. All in all, additional tooltip information would help increase the visualization's viability for helping developers choose from multiple refactoring candidates by adding them to the visualization first and deselecting them later.

The developers also had positive responses to the core features of the base JDeodorant tool, most notably its ability to implement the refactoring operations automatically. Comments regarding the user interface varied more widely. Some responded that they found the interface easy to use and understand, while others found it unintuitive, especially the button icons. The ability to sort the refactorings was suggested, so the user interface could be improved by implementing more features to make it easier for developers to quickly identify which refactorings they're looking for (such as a search feature).

4.5 Feature Ratings

The features the subjects were asked to rate on a scale of one to five, with five being the best, were the selection of multiple refactorings, the visualization of multiple refactorings, and the visualization of conflicting refactorings. These results can be seen in Table 3. All scored average ratings from between 4.125 and 4.5.

One of the most requested features was the ability to implement (and by extension, preview) all the selected refactoring operations at once. The developers acknowledged that visualizing their selected refactorings enabled them to refactor each one sequentially with confidence without having to run the identification feature after every application. This feature was introduced in the design phase of the extension's development, but was determined to be out of scope when we discovered there was no easy way to combine the operations into a single operation, rather than executing them automatically in sequence, which was deemed not significant enough of an improvement to warrant the necessary development time.

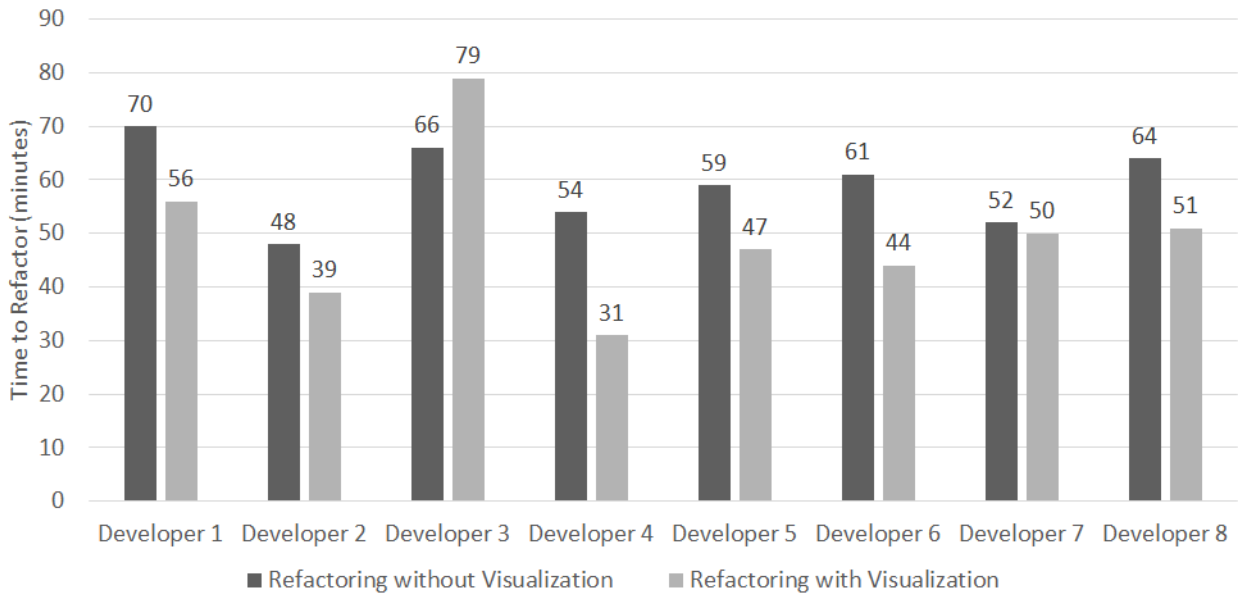

Figure 5: Refactoring Times per Developer

Table 3: Developer Ratings of Features (1:Bad, 5:Good)

Feature	Dev-1	Dev-2	Dev-3	Dev-4	Dev-5	Dev-6	Dev-7	Dev-8	Average Score
Selection of multiple refactorings	3	5	3	5	3	5	5	4	4.125
Visualization of multiple refactorings	5	5	3	5	5	4	4	5	4.5
Visualization of conflicting refactorings	5	4	4	4	4	5	4	4	4.25

4.6 Limitations of our extension

Naturally there are many languages aside from Java and its object-oriented cousins that utilize refactoring, and there are a multitude of development environments used for Java alone. While many aspects of the extension can be adapted for use in various environments with various languages, the technical aspects of the extension must remain limited to this specific language and environment. For example, many elements of the visualizations were immutable, such as the tooltips, and the code smell views lacked native support for checkboxes. Finding ways around these issues was not impossible, but restricted ideal and expedient development at times.

Additionally, the extension's design lacks input from a designer with experience in human-computer interaction, which could negatively impact its usability [5]. One of the facts that constrained the development of the extension was the use of built-in visualization tools in the Eclipse IDE, since JDeodorant is a plugin for that programming environment. The extension could undoubtedly be improved not only with expanded functionality, but with a stronger emphasis on making the extension easy to understand and use, and the visualizations clear and pleasing to the viewer. One improvement could be arranging the diagrams using a visually-appealing graphing algorithm, such as Delaunay triangulation [13], instead of a two-column grid.

5 THREATS TO VALIDITY

A potential threat relates to the experimentation itself. Despite having eight separate subjects to evaluate the tool, each developer has their own preferences and experiences that affect their ability to effectively utilize certain refactoring tools. These varying level of skills can affect their ability to use the tool and its effects, which is why we assigned each developer programs to refactor with and without the extension. This should help mitigate the learning curve and fatigue threat inherent to using the extension. Additionally, to mitigate the impact of a developer being particularly familiar or unfamiliar with refactorings performed on one of the projects, each developer was provided with two different projects. Furthermore, we instituted no time limit on the refactoring or questionnaire, and provided an instructional video walking the developer through using the tool and extension to refactor a sample project.

As the crux of this work deals with non-quantifiable concepts, having an accurate interpretation of human responses is a vital aspect of results analysis. Participants often have wildly varying and heterogeneous opinions. One of the ways we mitigated this threat to validity is by quantifying our results with a rating system. By having participants rate issues before responding to them, we can put together a rough indication of the developers' responses at a glance. However, this approach

does not allow for concrete statistical conclusions since developers normally interpret these ratings in their own way, with no universal reference. Additionally, this still limits the number of participants we can evaluate effectively, because of the necessity of performing a manual analysis and interpreting each participant's responses. Ultimately, we chose the participant and project sample sizes that focus on receiving clear feedback on the extension's function and its impact, rather than to definitively prove its statistical effectiveness for a given demographic.

6 CONCLUSION & FUTURE WORK

In this paper, we extended an existing refactoring tools, to better bridge the gap between refactorings and developers, through visualizing it. We address the limited transparency by providing developers with the possibility of verifying their refactoring outcomes. Practically, the extended tools provides timely visualization of multiple selected refactorings, and detects whether they may be conflicting. The extension encourages also the usage of various possible refactoring approaches through importing their exported refactorings.

The validation of the extension was performed through investigating whether it reduces refactoring effort, in terms of execution time. To do so, we convenience sampled eight developers, known to have participated in previous studies of refactorings, and we provided them with code smells to refactor. These code smells were detected using JDeodorant in 8 popular open-source tools. The feedback received shows promising results with respect to productivity and usability. Ultimately, this work proposes a novel approach to aid software developers in better understanding how to semi-automatically refactor their systems.

As future work, we would be ecstatic to see work continue in this vein of increasing trust between developer and tool, even if it is not related to our visualization. New types of visualizations, or new approaches altogether, will help to increase this topic's body of knowledge and eventually lead to at least one feasible solution. While our design went through several iterations, it does not mean that aspects deemed out of our scope could not be expanded upon in future projects and shown to be viable approaches for increasing a tool's level of trust.

REFERENCES

[1] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. IEEE, 51–58.

[2] Eman Abdullah Alomar. [n. d.]. Towards Better Understanding Developer Perception of Refactoring. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 624–628.

[3] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2019. On the impact of refactoring on the relationship between quality attributes and design metrics. In *International Symposium on Empirical Software Engineering and Measurement*. 1–11.

[4] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*. Springer, 387–419.

[5] Alan Dix. 2009. *Human-computer interaction*. Springer.

[6] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 18.

[7] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. 2007. Jdeodorant: Identification and removal of feature envy bad smells. In *International Conference on Software Maintenance*. 519–520.

[8] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 1037–1039.

[9] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology* 11, 2 (2012), 5–1.

[10] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[11] Almas Hamid, Muhammad Ilyas, Muhammad Hummayun, and Asad Nawaz. 2013. A comparative study on code smell detection tools. *International Journal of Advanced Science and Technology* 60 (2013), 25–32.

[12] Yoshio Kataoka, David Notkin, Michael D Ernst, and William G Griswold. 2001. Automated support for program refactoring using invariants. In *IEEE International Conference on Software Maintenance (ICSM'01)*. 736.

[13] Der-Tsai Lee and Bruce J Schachter. 1980. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer & Information Sciences* 9, 3 (1980), 219–242.

[14] Radu Marinescu. 2004. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 350–359.

[15] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. 1263–1270.

[16] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 331–336.

[17] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó'Cinnéide, and Kalyanmoy Deb. 2014. Software refactoring under uncertainty: a robust multi-objective approach. In *Annual Conference on Genetic and Evolutionary Computation*. 187–188.

[18] Mohamed Wiem Mkaouer, Marouane Kessentini, Mel Ó Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb. 2017. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering* 22, 2 (2017), 894–927.

[19] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. 2015. Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 3 (2015), 1–45.

[20] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. 2006. Does refactoring improve reusability?. In *International Conference on Software Reuse*. 287–297.

[21] Christian D Newman, Mohamed Wiem Mkaouer, Michael L Collard, and Jonathan I Maletic. 2018. A study on developer perception of transformation languages for refactoring. In *Proceedings of the 2nd International Workshop on Refactoring*. 34–41.

[22] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 3 (2016), 1–53.

[23] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and M. S. Hamdi. 2015. Improving multi-objective code-smells correction using development history. *Journal of Systems and Software* 105 (2015), 18–39.

[24] Anthony Peruma, Mohamed Wiem Mkaouer, Michael John Decker, and Christian Donald Newman. 2019. Contextualizing rename decisions using refactorings and commit messages. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 74–85.

[25] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84, 10 (2011), 1757–1782.

[26] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Ranking refactoring suggestions based on historical volatility. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 25–34.

[27] Eva Van Emden and Leon Moonen. 2002. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 97–106.