A Live Environment to Improve the Refactoring Experience

Sara Fernandes
Faculty of Engineering,
University of Porto
INESC TEC, Porto
Porto, Portugal
sfcf@fe.up.pt

Ademar Aguiar
Faculty of Engineering,
University of Porto
INESC TEC, Porto
Porto, Portugal
aaguiar@fe.up.pt

André Restivo
Faculty of Engineering,
University of Porto
LIACC, Porto
Porto, Portugal
arestivo@fe.up.pt

ABSTRACT

Refactoring helps improve the design of software systems, making them more understandable, readable, maintainable, cleaner, and self-explanatory. Many refactoring tools allow developers to select and execute the best refactorings for their code. However, most of them lack quick and continuous feedback, support, and guidance, leading to a poor refactoring experience. To fill this gap, we are researching ways to increase liveness in refactoring. Live Refactoring consists of continuously knowing, in real-time, what and why to refactor. To explore the concept of Live Refactoring and its main components — recommendation, visualization, and application, we prototyped a Live Refactoring Environment focused on the Extract Method refactoring. With it, developers can receive recommendations about the best refactoring options and have support to apply them automatically. This work helped us reinforce the hypothesis that early and continuous refactoring feedback helps to shorten the time needed to create high-quality systems.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; **Real-time systems software**; *Maintaining software*; • **Human-centered computing** → *Visualization techniques.*

KEYWORDS

code smells, refactoring, code quality metrics, software visualization, liveness

ACM Reference Format:
Sara Fernandes, Ademar Aguiar, and André Restivo. 2022. A Live Environment to Improve the Refactoring Experience. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming (‹Programming› '22 Companion), March 21–25, 2022, Porto, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3532512.3535222

1 INTRODUCTION

Current software systems tend to be large and complex. Reading, adapting, and maintaining them could be the most demanding and expensive tasks in the software development cycle because of possible hidden defects in the source code [2]. These issues can be named Code Smells, which represent surface indicators that something is wrong with the structure of the code [20].

To mitigate these problems, we can apply specific refactoring techniques to improve the code's readability, adaptability, and maintainability, leading to a cleaner and self-explanatory system [20]. There are several refactoring approaches and tools available [5, 9, 19, 48]. However, most of them show the blocks of code that should be refactored and how they do it, only when demanded by the developer, which sometimes can be too late in the development cycle, making the overall refactoring process harder and more expensive than it should be. Later feedback in software development can create complex and large systems, with more blocks of code to be analyzed, understood, and, consequently, refactored.

Several approaches have already tried to shorten the feedback loop between each programming action and its results to provide quicker guidance and support developers to converge to better source code faster [28]. Tanimoto [42, 43] focused part of his work on the "edit-compile-link-run" loop to improve the interaction between developers and their software systems, a concept called *Live Programming*.

Similarly, we are researching new ways of introducing liveness into the Refactoring process [20, 28]. *Live Refactoring* aims at improving the liveness of the refactoring-loop composed of three moments: the **identification**, **recommendation**, and **application** of refactoring to improve the software at hands. We consider it important to provide immediate and continuous feedback, support, and guidance to developers about possible refactoring candidates. Therefore, developers can become aware of what and how to improve their software while programming, thus creating more adaptable and maintainable code in earlier development stages.

To research more about Live Refactoring and its main activities and to verify if a live refactoring environment is capable of helping software developers achieve code with more quality and faster, we developed a prototype for Java and IntelliJ IDE. The main goal of this tool was to identify, recommend, and apply possible Extract Method refactoring candidates, using code quality metrics in real-time to help developers converge to better programming solutions faster and thus improve their refactoring experience. We focused on the Extract Method refactoring since it can solve frequent problems like long methods or duplicated code, and it can be part of many other refactorings. Also, we believe it is an essential refactoring because when we extract code to a new method, the original code becomes better organized, more understandable, and easier to adapt and maintain [20].

‹Programming› '22 Companion, March 21–25, 2022, Porto, Portugal

ACM ISBN 978-1-4503-9656-1/22/03…$15.00
https://doi.org/10.1145/3532512.3535222

This paper is organized into four sections. In Section 2, we present the background and related work on the main concepts involved. In Section 3, we detail our *Live Refactoring Environment* and its main components. In Section 4, we draw the main conclusions from our approach and summarize possible future work.

2 BACKGROUND AND RELATED WORK

In this section, we provide some background on the main topics of our research, namely: code smells, refactorings, code quality metrics, software visualization, and live programming. We also present some relevant examples of the current state-of-the-art.

2.1 Code Smells

Kent Beck introduced the concept of code smells in the late 90s. However, Martin Fowler made it gain more popularity when he talked about it in his book. Accordingly to both of them, a code smell is a surface indicator of a possible deeper problem in a software system. This kind of problem tends to decrease the readability, adaptability, and maintainability of the code [20].

The existence of code smells depends on several external factors. There are different smells for different programming languages, and their impact depends on their programming context. Most of them are inserted into the code by poor programming practices or by constant changes in the software requirements [38, 46].

Several tools and approaches aim to detect code smells. c-JRefRec is an Eclipse plugin that detects Feature Envy code smells, considering distinct heuristics [48]. WebScent finds embedded smells such as mixed code for HTML, CSS, or JavaScript, in web applications [33]. Palomba [35] proposed a textual approach to detect Long Method code smells. Stolee et al.[40] developed a tool that identifies smells on Yahoo! pipes. Fenske et al. [12] created FeatureIDE that detects three different types of code clones.

2.2 Refactoring

To mitigate a code smell, we need to refactor the code. Refactoring consists of restructuring the internal structure of software systems without changing their implemented behaviours. With it, we can improve code's readability, adaptability, and maintainability, making it cleaner and self-explanatory [9, 20].

To apply one specific refactoring to the code, we need to go through a set of several activities. First, we need to find code smells on the code. Then, we must identify refactoring candidates, considering the detected smells. After, we must choose the best refactorings to apply in that programming context. Lastly, we need to check if, after implementing a refactoring, the behaviours already implemented are preserved [31, 52].

We found several tools that identify possible refactoring opportunities. refactorix is a VSCode extension for JavaScript and TypeScript software systems that searches for Split and Extract Variable refactorings on code [34]. ExtC is an Eclipse plugin that detects possible Extract Class refactorings by measuring the cohesion of each class and by grouping class fields and methods by affinity [9]. Barbosaa [4] developed a plugin that uses clustering algorithms to find Extract Method refactoring opportunities. Vimala [49] and Bavota et al. [5] proposed an approach that uses game theory to identify the best refactorings to apply in a specific context to code.

2.3 Code Quality Metrics

Code quality metrics help developers to know more about the quality of their software systems and to predict, detect and identify possible problems, such as code smells and refactoring opportunities [3, 15, 17]. Metrics are standard values used to compute specific properties or attributes of the code as a way of assessing its quality [10, 24, 26].

JDeodorant is an Eclipse plugin that detects smells like Long Method, God Class, or Duplicated Code and suggests refactorings such as Extract Class or Extract Method. To do so, it uses clustering algorithms mixed with several metrics like the number of lines of code, coupling, or lack of cohesion [19]. Tech Debt Tracker is a plugin that assesses methods length, nesting depth, number of parameters, or density of comments of the source code [39]. Salgado [37] also developed a plugin that assesses several code quality metrics, such as lack of cohesion or cyclomatic complexity, as heuristics to identify Extract Variable, Extract Method, and Extract Class refactoring sequences. Tsantalis et al. [45] used the distance between entities to locate Move Method refactoring opportunities.

2.4 Software Visualization

Software visualization helps with the abstraction, representation, and perceptualization of software and its code using different visual techniques and metaphors. It enables developers to understand, change, and maintain their large and complex software systems quickly and easily, helping them to achieve better programming solutions faster [8, 25, 41].

We can find visual techniques being used in almost each code analysis tool. SeeSoft is a visualization tool that represents lines of code using colored pixels. The dimension of each pixel is proportional to the length of each line of code. Its color is assigned considering the structure of the code where the line is inserted [11]. Sv3D is a 3D visualization tool that draws different lines of code in a 3D space [8, 27]. Code City is a real-time software visualization tool that helps developers understand their code while programming. This tool uses the city metaphor to represent each part of the code [50]. Panas et al. [36] also used the city metaphor to quickly portray the architecture of software systems to help developers inspect their code quickly.

2.5 Live Programming

Live Programming is commonly understood as the act of programming systems while they are already executing code. Live Programming Environments enable software developers to be informed about the results of their programming actions while implementing them. Therefore, it helps developers shorten the "edit-compile-link-run" loop by providing possible visual and real-time feedback, support, and guidance about their software [1, 16]. With it, developers can create and explore code with high readability, adaptability, and maintainability and also evolve software faster [42, 43].

This concept gained prominence when Tanimoto [42] introduced a liveness hierarchy containing four levels (the first four levels of Figure 1): at Level 1, no semantic feedback about a program is provided; in Level 2, semantic feedback is available on-demand on a selected component; at Level 3, incremental semantic feedback is automatically provided with every program edit; on Level 4,

also incremental semantic feedback is automatically provided for other data events such as user events or exceptions. Years later, Tanimoto presented a new hierarchy that contained the four older levels and two new ones, as shown in Figure 1. Level 5 consisted of computers executing the code while developers were implementing it and foreseeing multiple next actions that could be implemented. Level 6 is related to computers predicting several next steps of developers and strategically selecting the best ones that should be implemented [43].

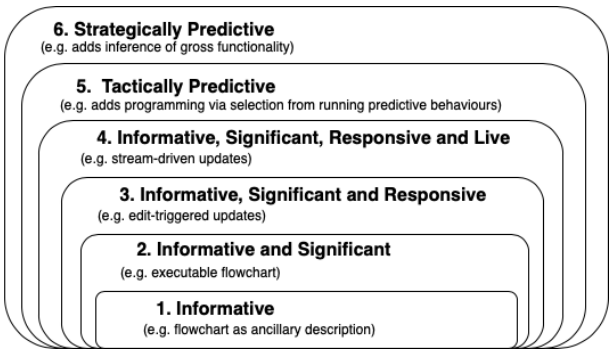

Figure 1: Liveness levels proposed by Tanimoto [43].

Circa is a live language and environment that enables developers to implement their software easily and quickly. With it, they can click on a specific part of the output and immediately inspect the block of code that implemented that behaviour or provided that result [18]. SuperGlue is a live reactive language that aims to create software based on components connected by signals, which are values of the data-flow model [29]. Finally, Roel [51] believed that there was a gap between the design and development of software caused by the lack of liveness in the development cycle. So, he created a framework that automatically assesses several conditions after any change is made to warn the developer about possible system failures.

2.6 Live Refactoring

As mentioned before, liveness helps developers implement software more easily and quickly. The main reasons that motivate the implementation of liveness on the software development are **immediacy**, **exploration**, and **stability**, enabling developers explore and inspect their code in real-time. Besides, they have more control over their systems because of the instant and continuous feedback on how they should improve and evolve their code [1]. Therefore, it might be useful to be able to benefit from liveness when refactoring software, since analogous to how live programming improves the feedback loop, live refactoring should improve the feedback cycle on the design of the code. Considering the strengths of liveness [1, 42, 43] and refactoring [20], we are researching how live refactoring may help developers to shorten the refactoring-loop. In concrete, how would help developers to build high-quality software systems faster since they would receive feedback, support, and guidance in earlier development stages while implementing their software [13, 14].

LambdaFicator is a live refactoring tool developed for Java expressions. It includes two different lambda-specific refactorings that aim to create cleaner code in just a few seconds [21]. Kobold is a live refactoring tool for web applications based on user interaction and change-history to detect code smells and suggest proper refactoring techniques [23]. Benefactor is an Eclipse live refactoring plugin that suggests refactorings techniques to developers when they try to refactor their code manually [22].

3 LIVE REFACTORING ENVIRONMENT

The effort and time needed to read, adapt, and maintain complex software systems are very high. And to make it worse, most developers try to refactor their code manually, which takes a lot of time [9, 32]

Current development environments already provide some assistance to developers. However, most of them are characterized by the lack of real-time feedback, support, and guidance [47], increasing the temporal distance between each refactoring implemented and the changes made to the code quality metrics.

To mitigate these problems, we propose a ***Live Refactoring Environment*** capable of inspecting code and measuring several code quality metrics to detect code smells and identify possible refactoring candidates, in real-time. With it, we aim to provide support and guidance to developers to help them achieve better programming solutions faster than with conventional refactoring tools.

The development of our *Live Refactoring Environment* was divided into several phases:

(1) **Selection of Proper Development Environment:** in this phase, we studied the best development environment to implement our approach;
(2) **Assessment of Code quality Metrics:** in this phase, we measured several code quality metrics to find specific code smells and identify the respective refactoring candidates;
(3) **Identification, Suggestion, and Application of Refactorings:** in this phase, we used the assessed metrics to detect code smells and to identify and suggest the refactorings that should mitigate each smell;
(4) **Application of Liveness:** in this phase, we implemented the live mechanisms needed to reduce the refactoring-loop;
(5) **Implementation of Visual Techniques:** in this phase, we implemented a visual technique that helped suggest the refactoring candidates to the developers.

3.1 Selection of Proper Development Environment

To start developing our refactoring approach, we had to choose which IDE we would use. First, we chose Visual Studio Code (VSCode), as it is an attractive modern environment widely used by software developers [6]. With this IDE, we started creating a Live Refactoring Environment for JavaScript and TypeScript software systems. VSCode provides access to a complete API that quickly analyzes and manipulates JavaScript and TypeScript code. Moreover, the API also offered several live and refactoring mechanisms that slightly help reduce the refactoring-loop.

With everything sketched, we started to develop our approach. However, over time, we realized that it would be challenging to test our environment since there aren't many large projects developed in JavaScript or TypeScript that have many code smells. Then, we re-examined our options and decided to migrate all the work developed so far to another IDE. In this case, we chose IntelliJ. As it happens with VSCode, it contains several APIs that allow us to analyze and manipulate code quite easily and several refactoring and live mechanisms that meet our goals. We had to change all the code until then developed in TypeScript to Java. After an extensive analysis of state-of-the-art, we found that Java is one of the most used programming languages to develop refactoring and metrics analysis tools. Besides, there are several large projects developed in Java that have been used in experiments carried out by other authors with similar objectives than ours [19, 44].

The migration of the project from VScode to IntelliJ didn't change the logic used either to measure the different quality metrics or to identify and apply the different refactoring opportunities.

3.2 Assessment of Code Quality Metrics

Before measuring specific code quality metrics, we had to select which metrics we wanted to analyze from a vast catalog of possible metrics. Hence, we chose more than twenty metrics from the number of lines of code to the Halstead complexity metrics, as can be seen in Table 1. These metrics help not only in the analysis of possible refactoring candidates but also in the assessment of the code quality.

Table 1: Summary of the code quality metrics supported by the proposed approach, grouped by type.

Type of Metric	Code Quality Metric
File Metrics	Number of Lines of Code, Number of Comments, Number of Classes, Number of Methods, Average Number of Long Methods, Average Lack of Cohesion, Average Cyclomatic Complexity
Class Metrics	Number of Fields, Number of Public Fields, Number of Methods, Number of Long Methods, Class Lack of Cohesion, Average Cyclomatic Complexity
Method Metrics	Number of Parameters, Number of Lines of Code, Number of Comments, Number of Statements, Method Lack of Cohesion, Cyclomatic Complexity, Halstead Length, Halstead Vocabulary, Halstead Volume, Halstead Difficulty, Halstead Effort, Halstead Level, Halstead Time, Halstead Bugs Belivered, Halstead Maintainability

To measure each of these metrics, we used the abstract syntax tree (AST) that represented the code focused on the editor at the time of the analysis. Both in VSCode and IntelliJ, it is possible to have access to ASTs quite easily since there is an API for that. Focusing now on IntelliJ, we obtained PsiTrees that represent the ASTs of the code under analysis, where each class is represented by a PsiClass, a method by a PsiMethod, and so on.

3.3 Identification, Suggestion, and Application of Refactorings

While measuring the code quality metrics, we started identifying possible refactoring candidates using the assessed values. Currently, our approach is only focused on detecting Extract Method refactorings.

The ***Extract Method*** refactoring consists of extracting a block of code from a method to a new one. The first step to find an Extract Method candidate is to select the fragments of code that could be extracted into a new method. First, we extract individual nodes from the methods of a class from a specific source file. Second, we combine each node found in the first step.

After finding all the possible extractable code fragments and combining them, our approach only considers specific cases, as can be seen in Algorithm 1 [37]. To be an Extract Method candidate, a block of code should be formed only by consecutive statements (line 17), have more than three statements (line 31), and it shouldn't contain more than 80% of the number of statements of the original method (line 32). These values can be changed in a proper configuration menu.

We consider specific metrics once we find the final set of Extract Method opportunities. This step of our approach is based on the methodology proposed by Meananeatra, Panita et al. [30] to evaluate Extract Method candidates. Their method evaluates three code quality metrics: the cyclomatic complexity **(CC)**, the number of lines of code **(LOC)**, and also the lack of cohesion **(LCOM)** of the candidate. By default, our development environment first tries to minimize the highest values of cyclomatic complexity, and after, it tries to reduce the number of statements and the lack of cohesion.

After identifying and sorting the refactoring candidates, we suggest them to the developer using a visual technique, which will be further explained in Section 3.5, and which presents each candidate through colors placed near each statement (Figure 2).

Each color is clickable. Suppose a developer clicks in a specific color. In that case, it will have access to a menu where he can check the refactoring candidates related to the statement painted by that color (Figure 3). Then, he can select the Extract Method that he aims to apply (e.g., he can decide if he wants to apply a refactoring from line 10 to 15 containing statement 1 or the refactoring from line 10 to 20 that includes the same statement).

After selecting the refactoring, it is applied automatically to the source code. Whether we were using VSCode or IntelliJ, this step is quite simple. Both have an API that allows us to perform specific refactorings on the source code, including the Extract Method refactoring, by only providing the elements of the candidate to be extracted.

When the refactoring is fully applied, our approach measures the code quality metrics before and after the changes performed on the code and saves them on a Firebase database. Then, our Live Refactoring Environment starts a new analysis of the new code to find new refactoring opportunities and so on.

```
public double[][] getJaccardDistanceMatrix(MyClass sourceClass) {
    ArrayList<Entity> entities = new ArrayList<>();
    entities.addAll(sourceClass.getAttributeList());
    entities.addAll(sourceClass.getMethodList());
    double[][] jaccardDistanceMatrix = new double[entities.size()][entities.size()];
```

Figure 2: An example of our Live Refactoring Environment presenting the refactoring candidates to its users.

Algorithm 1 Algorithm used to get the Extract Method candidates [37].

```
1: ranges ← []
2: nodes ← []
3: methods ← []
4: candidates ← []
5: aux ← []
6: for each fragment ∈ fragments do ▷ fragments of code selected from the first step of this approach
7:     ranges.add(fragment.range)
8:     nodes.add(fragment.nodes)
9:     methods.add(fragment.method)
10: end for
11: for i ← 0 to ranges.length do
12:     candidate ← Candidate(ranges[i], nodes[i], methods[i])
13:     aux.add(candidate)
14: end for
15: for i ← 0 to aux.length − 1 do
16:     for j ← 0 to aux.length do
17:         if areConsecutive(aux[i], aux[j]) then ▷ method that verifies if the nodes are consecutive or not
18:             range ← Range(aux[i].range.start, aux[j].range.end)
19:             nodes ← addAll(aux[i].nodes, aux[j].nodes)
20:             method ← aux[i].method
21:             candidate ← Candidate(range, nodes, method)
22:             if candidate not in aux then
23:                 aux.add(candidate)
24:             end if
25:         end if
26:     end for
27: end for
28: for each candidate ∈ aux do
29:     numStatements ← candidate.nodes.length
30:     oldStatements ← candidate.method.length
31:     if numStatements ≥ 3 then
32:         if numStatements ≤ 80% × oldStatements then
33:             candidates.add(candidate)
34:         end if
35:     end if
36: end for
```

3.4 Application of Liveness

After starting our Live Refactoring Environment, it assesses the current code quality metrics of the source code and visually presents the refactoring candidates to the developers. Then, they may or may not apply the suggested refactorings. If they prefer to apply them, they will be automatically applied to the code. After that, our approach starts a new process to analyze the new source code and find new refactoring candidates.

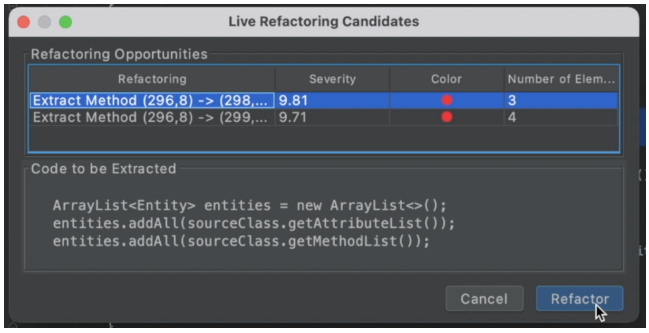

Figure 3: An example of our Refactoring Menu.

This is only possible through triggers provided by the IDE's API that initiate whenever specific actions are performed. VSCode and IntelliJ allow knowing when something changed in the code focused on the text editor. However, we consider that the triggers offered by IntelliJ are more complete, and they enable us to perform our process at the right moments and not at every character changed as it happens with VSCode. Taking this in mind and focusing on IntelliJ as the IDE that we chose to develop our approach, we used three different triggers. Two of them to initialize the analysis process before and after each refactoring applied on the code or when a block of code is manually changed (Algorithm 2), and another one to initialize the same process when we switch focus between text files on the text editor. On the second trigger, to shorten the refactoring-loop (Algorithm 2), we only measure the new metrics related to methods involved in the refactoring performed and not all the metrics from all the existing methods of that file.

Therefore, our approach is very reactive to changes made in the code and in the text editor itself, providing quick feedback, support, and guidance to the developers about the code they are working on.

3.5 Implementation of Visual Techniques

After sorting the identified refactorings, we calculate the severity of each one of them, taking into account their descending order (the first refactoring candidate should be the one with higher severity) and the number of candidates detected. Then, each value is normalized on a scale between 1 to 10. We chose this scale because of

Algorithm 2 Triggers used to initiate the analysis of possible refactoring candidates when a change is made on the source code.

```
1:  sourceFile ← editor.getCurrentFile()
2:  metricsBefore ← null
3:  metricsAfter ← null
4:  refactorings ← []
5:  if before then
6:      metricsBefore ← FileMetrics(sourceFile)
7:  else if after then
8:      metricsAfter ← before.changeMetrics(refactoring) ▷ only changes the metrics related to the methods involved on the refactoring performed
9:      newSourceFile ← editor.getCurrentFile()
10:     refactorings ← calculateRefactorings(newSourceFile) ▷ find the new refactoring in the new source code
11: end if
```

the color gradient used to represent each candidate. Our gradient is composed of ten colors from bright green to dark red, as can be seen in Figure 4, where green represents code that doesn't need to be immediately refactored, and red represents code that needs to be extracted with urgency. Therefore, the severity of each candidate is the index of each color on the gradient.

Figure 4: Color gradient used to represent each refactoring candidate.

Each color is then painted on the left side of the text editor (Figure 2), and developers can click on them to have access to the refactoring menu, where they can choose the most correct refactoring to be applied (Figure 3).

This visual technique was created using the IntelliJ API that allows the implementation of colored gutters on the text editor. Each gutter is represented by a specific position on the source code, an icon created through PNG images, which in our case represents each color from the color gradient, and also by a trigger that is activated when clicking on it. This trigger is used to activate the refactoring menu. Then, after having all the gutters initialized, we implement a highlighter that allows placing each gutter on the corresponding position on the left side of the focused text editor.

3.6 Limitations

Currently, our Live Refactoring Environment still has several limitations. One of them is the small number of refactoring techniques included. Currently, we are only focused on detecting Extract Method opportunities. We consider it a limitation because identifying only one type of refactorings decreases the range of code smells that can be mitigated through code, reducing the opportunities to improve its quality.

Another limitation of our solution is the visual representation of each refactoring opportunity on the source code. Since we haven't carried out any usability experiment, we don't know if our approach is intrusive or not, distracting software developers from their programming actions and decreasing their programming experience.

Finally, another limitation is the total time needed to provide feedback, support, and guidance about the possible refactorings that should be applied to the code. Currently, our Live Refactoring Environment takes a few seconds to present all the refactoring opportunities to its users, which in our opinion, should be under one second, being thus too long for a live approach. In this sense, it is a topic that must be taken into account, and that must be improved in the future.

4 CONCLUSIONS

Software systems can be hard to read, understand, adapt, and maintain [2]. To reduce these problems, programmers frequently try to inspect their source code to detect possible defects or problems like code smells and, therefore, possible refactoring opportunities. However, it could be hard to identify refactoring candidates in the middle of complex code with low readability [20].

We research ways to augment an environment to be able to identify, suggest, and apply refactorings on the source code, in real-time, to reduce the effort and time needed to modify the code, and improve its ability to understand, change, and maintain it. We claim, but do not prove in this paper, that having these suggestions in earlier development stages would help developers create code with more quality and faster.

Taking this in mind, we proposed a *Live Refactoring Environment* that positively improves the feedback, support, and guidance provided to developers as possible refactoring suggestions. To do so, it assesses several code quality metrics that help detect and sort blocks of code that should be refactored. It also visually identifies each refactoring candidate on the source code, in real-time, while programming, to raise the developers' awareness on the flaws present in their code. With this quick and constant feedback, we believe that developers would be able to reduce the time needed to implement high-quality software systems.

4.1 Future Work

In the future, we expect to detect more code smells such as Lazy Class, God Class, Feature Envy, Shotgun Surgery, or Duplicated Code. To solve each new code smell, we also aim to introduce new refactorings like Extract Class, Extract Variable, Move Method, or Inline Class [20]. To improve this topic, we will probably need to measure more code quality metrics like cognitive complexity [7] and use different approaches to identify new refactoring opportunities on the source code. We also plan to implement smart algorithms to find the best refactoring sequences that should be applied in a specific programming context. Besides, we want to correct and improve the level of liveness of our refactoring environment to reduce the time needed to provide visual feedback, support, and guidance to its users.

Finally, after implementing our solution, we plan to execute an empirical experiment that will help us validate our solution and assess the development experience provided by our Live Refactoring Environment. With it, we will verify whether developers could

quickly improve their attitude towards better code or not, and how faster.

ACKNOWLEDGMENTS

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project 2020.05161.BD.

REFERENCES

[1] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. 2019. Live Software Development: Tightening the Feedback Loops. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming* (Genova, Italy) *(Programming '19)*. Association for Computing Machinery, New York, NY, USA, Article 22, 6 pages. https://doi.org/10.1145/3328433.3328456

[2] Rajiv D Banker, Srikant M Datar, Chris F Kemerer, and Dani Zweig. 1993. Software complexity and maintenance costs. *Commun. ACM* 36, 11 (1993), 81–95.

[3] J. Bansiya and C.G. Davis. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering* 28, 1 (jan 2002), 4–17. https://doi.org/10.1109/32.979986

[4] João Paulo Moreira Barbosa. 2020. *Towards a Smart Recommender for Code Refactoring.* Master's thesis. Faculty of Engineering of the University of Porto, Porto, Portugal.

[5] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Giuliano Antoniol, and Yann-Gael Gueheneuc. 2010. Playing with Refactoring: Identifying Extract Class Opportunities through Game Theory. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, USA, 1–5. https://doi.org/10.1109/ICSM.2010.5609739

[6] Atul Bhatt. 2021. Visual studio V/s vs code?: Ide or editor?: Find out what you need!: Make an informed choice. https://medium.com/analytics-vidhya/visual-studio-v-s-vs-code-ide-or-editor-find-out-what-you-need-make-an-informed-choice-5bb2a4f48ec2

[7] G. Ann Campbell. 2018. Cognitive Complexity: An Overview and Evaluation. In *Proceedings of the 2018 International Conference on Technical Debt* (Gothenburg, Sweden) *(TechDebt '18)*. Association for Computing Machinery, New York, NY, USA, 57–58. https://doi.org/10.1145/3194164.3194186

[8] Pierre Caserta and Olivier Zendra. 2011. Visualization of the Static Aspects of Software: A Survey. *IEEE Transactions on Visualization and Computer Graphics* 17, 7 (jul 2011), 913–933. https://doi.org/10.1109/TVCG.2010.110

[9] Keith Cassell, Craig Anslow, Lindsay Groves, and Peter Andreae. 2011. Visualizing the refactoring of classes via clustering. *Conferences in Research and Practice in Information Technology Series* 113, Acsc (2011), 63–72.

[10] S.R. Chidamber and C.F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (jun 1994), 476–493. https://doi.org/10.1109/32.295895

[11] S. C. Eick, J. L. Steffen, and E. E. Sumner. 1992. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering* 18, 11 (1992), 957–968.

[12] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. 2017. Variant-preserving refactorings for migrating cloned products to a product line. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, USA, 316–326. https://doi.org/10.1109/SANER.2017.7884632

[13] Sara Fernandes. 2021. A Live Environment for Inspection and Refactoring of Software Systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1655–1659. https://doi.org/10.1145/3468264.3473100

[14] Sara Fernandes, Ademar Aguiar, and André Restivo. 2020. Live Software Inspection and Refactoring. In *8th SEDES, Software Engineering Doctoral Symposium*. CEUR Workshop Proceedings, Germany, 1–10.

[15] Sara Fernandes, André Restivo, Hugo Sereno Ferreira, and Ademar Aguiar. 2020. Helping Software Developers through Live Software Metrics Visualization. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *('20)*. Association for Computing Machinery, New York, NY, USA, 209–210. https://doi.org/10.1145/3397537.3397539

[16] Sara Fernandes, André Restivo, Hugo Sereno Ferreira, and Ademar Aguiar. 2020. Helping Software Developers through Live Software Metrics Visualization. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *('20)*. Association for Computing Machinery, New York, NY, USA, 209–210. https://doi.org/10.1145/3397537.3397539

[17] Sara Filipa Couto Fernandes. 2019. *Supporting Software Development through Live Metrics Visualization.* Master's thesis. Faculty of Engineering of the University of Porto, Porto, Portugal.

[18] Andrew Fischer. 2013. Introducing circa: A dataflow-based language for live coding. *2013 1st International Workshop on Live Programming, LIVE 2013 - Proceedings* 1 (2013), 5–8. https://doi.org/10.1109/LIVE.2013.6617339

[19] 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* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1037–1039. https://doi.org/10.1145/1985793.1985989

[20] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code (2nd Edition).* Addison-Wesley, Boston, USA.

[21] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. 2013. LAMBDAFICATOR: From Imperative to Functional Programming through Automated Refactoring. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, USA, 1287–1290.

[22] Xi Ge and Emerson Murphy-Hill. 2011. BeneFactor: A Flexible Refactoring Tool for Eclipse. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 19–20. https://doi.org/10.1145/2048147.2048157

[23] Julián Grigera, Juan Cruz Gardey, Alejandra Garrido, and Gustavo Rossi. 2018. Live versioning of web applications through refactoring. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*. ACM Press, New York, New York, USA, 872–875. https://doi.org/10.1145/3238147.3240483

[24] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series).* Elsevier Science Inc., USA.

[25] István Kádár, Péter Hegedundefineds, Rudolf Ferenc, and Tibor Gyimóthy. 2016. A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering* (Ciudad Real, Spain) *(PROMISE 2016)*. Association for Computing Machinery, New York, NY, USA, Article 10, 4 pages. https://doi.org/10.1145/2972958.2972962

[26] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. 2006. *Object-Oriented Metrics in Practice.* Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/3-540-39538-5

[27] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 2003. 3D Representations for Software Visualization. In *Proceedings of the 2003 ACM Symposium on Software Visualization* (San Diego, California) *(SoftVis '03)*. Association for Computing Machinery, New York, NY, USA, 27–ff. https://doi.org/10.1145/774833.774837

[28] Robert Cecil Martin. 2003. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall PTR, USA.

[29] Sean McDirmid. 2007. Living It up with a Live Programming Language. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 623–638. https://doi.org/10.1145/1297027.1297073

[30] Panita Meananeatra. 2012. Identifying Refactoring Sequences for Improving Software Maintainability. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. Association for Computing Machinery, New York, NY, USA, 406–409. https://doi.org/10.1145/2351676.2351760

[31] Tom Mens and T. Tourwe. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 2 (feb 2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817

[32] Emerson Murphy-Hill and Andrew P. Black. 2010. An Interactive Ambient Visualization for Code Smells. In *Proceedings of the 5th International Symposium on Software Visualization* (Salt Lake City, Utah, USA) *(SOFTVIS '10)*. Association for Computing Machinery, New York, NY, USA, 5–14. https://doi.org/10.1145/1879211.1879216

[33] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2012. Detection of Embedded Code Smells in Dynamic Web Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. Association for Computing Machinery, New York, NY, USA, 282–285. https://doi.org/10.1145/2351676.2351724

[34] Christian Oetterli. 2017. refactorix. https://marketplace.visualstudio.com/items?itemName=christianoetterli.refactorix.

[35] Fabio Palomba. 2015. Textual Analysis for Code Smell Detection. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2* (Florence, Italy) *(ICSE '15)*. IEEE Press, USA, 769–771.

[36] Thomas Panas, Dan Quinlan, and Richard Vuduc. 2007. Tool Support for Inspecting the Code Quality of HPC Applications. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing Applications (SE-HPC '07)*. IEEE Computer Society, USA, 2. https://doi.org/10.1109/SE-HPC.2007.8

[37] Sérgio António Dias Salgado. 2020. *Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics.* Master's thesis. Faculty of Engineering of the University of Porto, Porto, Portugal.

[38] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (apr 2018), 158–173. https://doi.org/10.1016/j.jss.2017.12.034

[39] Stepsize. 2021. Tech Debt Tracker. https://marketplace.visualstudio.com/items?itemName=Stepsize.tech-debt-tracker.

[40] Kathryn T. Stolee and Sebastian Elbaum. 2011. Refactoring Pipe-like Mashups for End-User Programmers. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 81–90. https://doi.org/10.1145/1985793.1985805

[41] M.-A.D. Storey, K. Wong, and H.A. Müller. 2000. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming* 36, 2-3 (mar 2000), 183–207. https://doi.org/10.1016/S0167-6423(99)00036-2

[42] Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing* 1, 2 (1990), 127 – 139. https://doi.org/10.1016/S1045-926X(05)80012-6

[43] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming* (San Francisco, California) *(LIVE '13)*. IEEE Press, Piscataway, NJ, USA, 31–34. http://dl.acm.org/citation.cfm?id=2662726.2662735

[44] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, USA, 4–14. https://doi.org/10.1109/SANER.2018.8330192

[45] N. Tsantalis and A. Chatzigeorgiou. 2009. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering* 35, 3 (may 2009), 347–367. https://doi.org/10.1109/TSE.2009.1

[46] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, USA, 403–414. https://doi.org/10.1109/ICSE.2015.59

[47] Yuriy Tynchuk. 2017. *Quality Aware Tooling*. Ph. D. Dissertation. University of Bern.

[48] Naoya Ujihara, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2017. c-JRefRec: Change-based identification of Move Method refactoring opportunities. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, USA, 482–486. https://doi.org/10.1109/SANER.2017.7884658

[49] S. Vimala, H. Khanna Nehemiah, R.S. Bhuvaneswa, G. Saranya, and A. Kannan. 2012. Applying Game Theory to Restructure PL/SQL Code. *International Journal of Soft Computing* 7, 6 (jun 2012), 264–270. https://doi.org/10.3923/ijscomp.2012.264.270

[50] Richard Wettel, Michele Lanza, and Romain Robbes. 2011. Software Systems as Cities: A Controlled Experiment. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 551–560. https://doi.org/10.1145/1985793.1985868

[51] Roel Wuyts. 2002. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. Ph. D. Dissertation. Vrije Universiteit Brussel, Faculteit Wetenschappen - Departement Informatica.

[52] N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue. 2016. Revisiting the relationship between code smells and refactoring. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, USA, 1–4.