Code Context based Generation of *Refactoring Guidance*

Student	Patrick de Beer
Student number	850327959

Date of Presentation	28th February 2019

This page is intentionally left blank.

Graduation Assignment

Code Context based Generation of *Refactoring Guidance*

Open University of the Netherlands
Faculty of Management, Science and Technology
Master's Program in Software Engineering

Student	*Dhr. Ing. Patrick de Beer*
Student number	*850327959*
Chairman	*Mw. Dr. Ir. Sylvia Stuurman*
Primary Supervisor	*Dhr. Dr. Ir. Harrie Passier*
Secondary Supervisor	*Dhr. Em. Prof. Dr. Lex Bijlsma*
Defense Date	*28th February 2019*
Course code	*IM9906*

Summary

Existing work shows us that the importance of high code quality is not naturally present in the mindset of undergraduate software engineering students. Emphasizing the importance of high quality software and teaching students some practices how it can be achieved, might prove useful. An example of such a practice, is software refactoring. Prior studies show that this practice is often mastered by learning on the job from peer experts. A situation that is not easily achieved in an educational setting. Intelligent tutoring systems could be an option to teach software refactoring. However, the availability of such systems, in the specific domain of software refactoring, is scarce.

In this study, we envision functionalities that such a tutoring system in the domain of software refactoring might provide. Together with these functionalities, we mention existing work that can prove useful in a future implementation. This summary of existing work shows that there is, to-date, a knowledge gap on how to automatically generate refactoring instructions for arbitrary compilable Java code. Specifically, on the refactorings described by Fowler.

In this study, we present an answer to this lack of knowledge by explaining how to automatically generate refactoring instructions for arbitrary compilable Java code based on code context. Our goal is to provide a student with instructions that are used manually refactor a piece of Java code, for a refactoring chosen by the student. This work is not generating instructions based on model solutions, but solely on code context. We are not guiding a student to one correct final solution, but provide hints and instructions with the intention to leave room for exploration, improve understanding of the refactoring process and make students aware of potential risks when refactoring the selected code.

A theory is presented explaining how to generate refactoring instructions based on the code context of a selected piece of Java code to be refactored. The theory consists out of two parts. First, a model that holds code contexts and related instructions for a selected refactoring. Second, an algorithm that is able to generate instructions, given: a concrete model, arbitrary Java code to manually refactor, and its code context. This theory is evaluated by a software prototype and two concrete models. These models have been created based on expert knowledge available for refactorings: *rename method* and *extract method*. The prototype demonstrates us that we can generate refactoring instructions out of Java code, where these instructions match with the code context of the code being refactored.

The generated results of our prototype are evaluated on a small-scale with third year undergraduate software engineering students in semi-structured individual and group interviews. The goal of this evaluation was to have early involvement of students and get additional insights. These interviews show us promising results when looking at how useful the students judge the generated refactoring guidance.

This study demonstrates that refactoring instructions on how to manually refactor an arbitrary piece of Java code can be generated and that students judge the generated outcome of the prototype and its concept as being useful. Still many aspects are open for future research before the envisioned tool in this study is a reality. Some examples for future work are: Extension of the used models, increased understanding how more complex scenarios are refactored according to expert knowledge and simplifications of the presented theoretical model.

List of Abbreviations Used

We abbreviate some regularly used concepts and terms in our text. All unfamiliar concepts and terms are explained subsequently in the theoretical section.

AST	*Abstract syntax tree*	**CCPD**	*Code context property detector*
CCA	*Code context advice*	**I**	*Instantiator*
CCP	*Code context property*	**RAG**	*Refactoring advice graph*

List of Figures

Index

1 Introduction

There is almost no activity in our daily lives that is not dependent on software somehow. Jones and Bonsignour estimate that in 2011 at least 50% of the population in the US was dependent on approximately 76 million lines of code for daily activities involving their PCs, media devices, automobiles, household appliances and web systems [26].

In their book, they also show us that these systems require an enormous effort to maintain because of lack of sufficient software quality. By their estimation, of 2,500,000 software professionals working in the U.S. in 2011, 1,000,000 of them were involved in fixing post-release bug issues.

Our growing dependency on software, the increasing size of software systems and the related growth of software complexity [24] will most certainly be a trend that continues in the near future. If we do not focus more on the quality of our software systems, the costs of maintenance will probably keep increasing together with the growth of software complexity [12, 26]. Besides the economic impact this will have, the impact of the growing number of software defects on society may also become larger and may be felt more frequently than the examples we have seen in past years.

These are some of the reasons that have led us to the belief that practices resulting in improved software quality should receive more attention in the education of future software engineers. Eventually, a focus on software quality does result in fewer defects [26].

Several studies show that there is much to gain in improving the quality mindset of students. There is not a clear improvement seen in software quality when the work of novice students is compared to that of upper-level students [9]. Code quality issues in students' software are left unchanged [32]. Students evaluate the correct working of software mainly from a functional perspective [35], and the main focus of students seems to be more on coding rather than on proper software design [54]. Our own observations of undergraduate software engineering students confirm the results presented in these earlier studies. Also, regular feedback from internships addresses issues such as maintainability, readability and software design. Unfortunately, it seems that we are not at the point where we can say that undergraduate software engineering students have sufficient knowledge and skills to develop high-quality software.

One goal of our study and possible future work is exploring possible solutions for achieving a better software quality mindset in students by offering tooling that can guide them in their code writing process. We think that students may develop better abilities to write maintainable and readable code if they are given the opportunity to experiment with and explore software refactoring. We see this as a promise of refactoring*: "Code will be read & modified more frequently than it will be written. The key to keep code readable and modifiable is refactoring"* [19]. A positive side effect would be that specialized training in refactoring would also prove useful in Agile development, which is now a common software development approach [22]. Refactoring is assumed to be part of the Agile development process [59].

We are not alone in suggesting more investment in refactoring skills. A comparison of the 2004 and 2014 ACM/IEEE curriculum guidelines for undergraduate software engineering students reveals that refactoring has been given a more prominent place [4, 36]. Several recent studies also emphasize the importance of including refactoring in intelligent tutoring systems [33, 63].

Although some tutoring systems exist that address software code quality improvement [14, 39], there are to the best of our knowledge none that guide students in performing the specific, more complex, refactorings as described by Fowler.

Within this study we look specifically at two refactorings described by Fowler: *rename method* and *extract method*. Our goal is that, for any piece of compilable Java code, we generate textual guidance on how to manually perform a refactoring on a piece of code. Students can select the refactoring and the code they want to refactor by themselves. This leads us to search for an answer to the following question:

> *How can we automatically generate refactoring guidance for Java code that is based on code context?*

The research question and our research approach are elaborated on in Chapter 4.

Our study contributes the following artifacts:

- A vision that suggests a workflow for an educational refactoring guidance tool. For each step in the workflow, references are included to existing work that can contribute to future implementations or to existing gaps in knowledge. This vision also has been used to define functional and non-functional aspects of the prototype we have developed to support our study.
- A theoretical model and algorithm for generating stepwise guidance related to manual refactoring of Java code that is based on code context.
- A prototype that demonstrates the working of our theory based on concrete models for the *rename method* and *extract method*.

The remainder of the paper follows this structure: Chapter 2 gives an overview of our literature study into several aspects of refactoring and the presence of educational support for it, such as intelligent tutor systems. Chapter 3 elaborates on the workflow we envision within an educational refactoring guidance tool. We use this vision as a source to formulate our research scope. Chapter 4 explains our research approach. Chapter 5 introduces and explains our theory on how to automatically generate stepwise guidance based on code context. The chapter is concluded with examples how our theory was translated to our prototype. Chapter 6 presents our gathered preliminary results: concrete models, results of an evaluation of our prototype, and student evaluation results. Chapter 7 discusses the preliminary results and suggests future work that may be relevant in our research context. Chapter 8 formulates a conclusion towards the research questions.

2 Theoretical Background

This chapter gives an overview of the performed literature study to get an understanding of existing work. We researched refactoring from multiple perspectives: the theory, the practical issues encountered in the field and the tooling currently available to assist with code refactoring. An important goal was to obtain a view on the problems that are often encountered when refactoring code and which existing tools can assist in refactoring code. Because this study aims at students, we also searched for tutoring systems that might be able to support them with refactoring their code.

2.1 Software Refactoring – a Practical Perspective

Refactoring is "*the process of changing a software system in such a way that it doesn't alter the external behavior of the code, yet it improves its internal structures*" [19]. The advantage of improving internal code structures is, according to Fowler, to maintain a clean software design and making code easier to understand. These improvements help in improving software quality aspects such as maintainability and understandability.

Fowler describes his expert knowledge of refactoring Java code extensively. Detailed descriptions are given for 72 refactorings. These descriptions consist of a main refactor scenario and points of attention for constructs in code that should be solved by following alternative refactoring approaches. The refactorings are coupled to 22 *code smells*. These are common code constructs that are suspected to cause software quality issues. For each code smell, it is suggested which refactoring might be suitable to improve the design of the code.

Several studies have looked at the relation of refactoring to software quality. Some of these studies have confirmed that refactoring can improve software quality when Fowler's refactoring procedures [30, 57] are strictly followed. It has to be noted, that these studies were run in a controlled lab environment.

Interestingly, studies that looked at the empirical effect of refactoring do not come with conclusive evidence that refactoring attempts always result in a positive effect on software quality [29, 64]. A possible explanation for this mismatch may be that in the professional field refactoring is not always applied properly. Fowler gives a clear warning that "*refactoring is risky*" and when not followed as a strict process can result in decreased code quality: "*Refactoring if not done properly, can set you back days, even weeks*" [19].

This interesting conflict has led us to the question of whether there are indicators that refactoring in practice may not be well enough understood or executed. Several studies have investigated how refactoring is applied and perceived by software engineers [34, 44, 60].

Refactoring is a complex activity, in which thorough analysis and understanding of code is necessary before changes are made in a disciplined way [15]. Several studies demonstrate that in practice this discipline is not always followed [44]. Refactoring in an undisciplined fashion is what Murphy-Hill calls *floss refactoring*. In this process the refactoring activity is intertwined with other software activities that do not guarantee the behavior-preservation of code—for example, adding new small functionalities while refactoring. This has also been observed by Kim: "*While refactoring is defined as a behavior-preserving code transformation in the academic literature, the de-facto definition of refactoring in practice seems to be very different from such a rigorous definition*" [34].

Besides not following a strict process for refactoring, developers lack a proper understanding of refactoring, according to some indicators. In a study into the use of automatic refactoring, developers often were not aware of which refactorings were available, and when they did know which were available, they could not always tell what these refactorings actually did [60]. Another study found that even when a code smell was given, developers did not always know which refactorings might prove useful for it [17]. Yet another study analyzed 12,922 refactorings and came to the conclusion that only 7% of the performed refactorings were actually removing a code smell [5] .

Based on these study results, we can say that there is at least room for improvement in how refactoring is applied in practice by software engineers. We leave it as an interesting question for future research whether there is a relationship between the lack of knowledge about refactoring and the lack of empirical results showing that refactoring indeed improves software quality.

2.2 Software Refactor Tools

Several tools were developed over the last two decades that can assist in refactoring Java code. We have tried to determine whether these tools can be directly useful in software engineering education and, if not, what in our opinion their shortcomings are. We discuss in more detail which properties of an educational refactoring tool we consider important in chapter 3.

Popular integrated development environments (IDEs) like IntelliJ [25] and Eclipse [23] do offer a set of automatic refactorings out of the box. In addition to these tools, many other stand-alone or plug-in tools have been developed as a result of research in the field of refactoring. A recent systematic literature review [51] shows us we can categorize tools into those that

1. Detect code smells
2. Generate diagrams such as class diagrams to visualize identified code smells
3. Perform automatic refactorings
4. Assess refactorings to determine whether they introduce side effects

In our case we have looked particularly into tools that perform automatic refactorings. We think that this category of tools best fits our goal of helping students improve their skills in the process of software refactoring.

Despite the large number of available tools, several studies have shown that many developers refactor most of their code manually [44, 60]. A more recent study comes to more or less the same conclusion: In this study, 86% of the developers said they do refactorings manually, and 50% even indicated they refactor all of their code in this fashion [34].

The reasons why developers do not use automatic refactor tools have been studied. In interviews with software developers, they indicate that refactorings make too many changes at once, which leads to overwhelming results. The results of a refactoring are seen as unpredictable because there is no clear understanding of what has been changed in the code [60]. Other studies point to reasons that are more related to the usability, reliability and correctness of the tools, suggesting that the tools contain major bugs, do a poor job of communicating errors or provide useless hints [44]. That we cannot fully rely on automatic refactorings is also mentioned in a study meant to make current automatic refactorings safer: *“Most of the refactoring tools do not implement all preconditions that guarantee the refactoring*

correctness, since formally identifying them is cost-prohibitive. Therefore, these tools may perform non-behavior preserving transformations" [53].

These shortcomings have been addressed in recent years, but still there is room for improvement. A recent review study of research published since 2004 gives a set of recommendations for future work [1]. One improvement suggested by Abebe is that tools should offer multiple solution paths to solve a code smell or that refactor solutions should not be fixed but be adjustable by the engineer. Most automatic refactoring tools lack these features and present us with a single solution only.

The suggestion for multiple and more flexible solution paths is also in line with a recommendation from another study that, because of the complexity of many refactorings, an analytic approach is needed. Such an approach would evaluate several solutions and make choices based on which solution offers the best improvable code. "*Refactoring is not yet to the point where we can solely depend on the tools. It is not just depending on simple transformation but on thorough analysis of source code"* [15]. Abebe also suggests that automatic refactoring should lead to more suitable or correct solutions, and that more of the total 72 possible refactorings mentioned by Fowler should be automated. Eclipse offers 23 refactorings out of the 72. IntelliJ offers 27 automatic refactorings.

With a small example, we can demonstrate that some of the issues Abebe mentioned can easily be observed in the latest refactoring tools. In the code fragment below, we wanted to automatically extract lines of code starting with the line marked "Extract from:" and ending with the line marked "Extract to". In this example we used the built-in refactor tools of two popular IDEs—IntelliJ 2018.1 and Eclipse Kepler SR2. We expected them to offer a solution that would still give correct values of variables *a* and *b* later on in the *OriginMethod* method.

```
Public void OriginMethod(){
    int a,b = 0;

    b++;          // Extract from:
    a++;          // Extract to:

    System.out.println(a+b);
}
```

Eclipse presented the message that *extract method* was not possible. In this case a *multiple solution path* could have added value by presenting alternative solutions to consider. Eclipse, however, stuck to the solution path that led to the message that it can extract only code that has exactly one changing variable that is used later in the code.

IntelliJ did slightly better by suggesting an alternative path. The proposal was to use the refactoring *Replace Method with Method Object*.

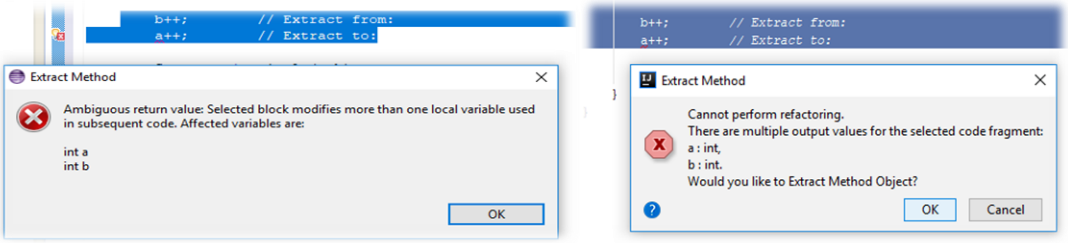


Figure 1 Result of Extract Method. Left: Eclipse refuses. Right: IntelliJ also indicates it cannot perform the action and offers a default solution with Extract Method Object

From a usability aspect we can question whether the choice "OK" should be made the default in the user interface. The reaction of an average student using the tool would probably be just to press Enter. In that case, the code transformation performed would be correct but not really more readable. The student likely would have been better off leaving the code unchanged. Given this case, we can reason that more alternative solutions should have been presented and that code transformations should result in a readable and maintainable improvement.

Having looked at how two current IDEs handled refactoring of the simple code example above, we next look at how this example can tell us about the usefulness of available refactoring tools in an educational context. Refactoring of simple scenarios is not trivial for automatic refactor tools. The presented solutions are not complete and refactoring outcomes might differ between tools, so it is not useful to train students in the use of one specific tool. From an explorative perspective, students using only a single tool could not analyze multiple solutions and reason about where risk might be present. Also, being presented with only one or two solutions could lead students to think that those were the only solutions available. Students also would not be able to develop an understanding of the steps that led to the final result, because those steps would not be visible. The tools offer a preview of the suggested code transformation, but this is the end result of a black-box approach. We can also interpret these previews as an indicator given by the manufacturers that automatic refactoring cannot be done blindly and that proper analysis must be done before transformations can be accepted. But this is not what we can expect from students when they do not have sufficient knowledge of the refactoring process that led to the presented result.

One academic tool that gave us insight into the steps it takes in performing a selected refactoring process is JSpirit. This tool can detect a number of code smells and offers the option of performing an automatic refactoring to clean up a smell. In preview mode, not only the end result, but also the code transformation steps are shown. Nonetheless, in this case we still see some shortcomings. Refactorings can only be performed on code smells detected by the tool, but the code smell detection seems rather conservative to prevent false positives. Also, the refactoring solutions presented offer only a single approach, so students would not be executing the refactoring themselves, step by step. We think that students' actually doing and solving the problem helps improve their understanding of the refactoring process. Students see where changes have been made in their code and experience the effects of those changes, which helps them understand or even experience the risks that might be related to the changes.

In summary, we can say that improvements have been made in the field of automatic refactoring tools, but there are still many issues remaining, which means that refactoring is not simply one click on a

button. It is even questionable whether this will be ever the case. Complex refactorings can result in many possible end solutions. These cases require engineers who understand refactoring and the impact of their choices very well. Current tooling typically offers only one solution, and the process seems not transparent enough to allow for proper decisions as to what the impact of refactoring changes will be. Finally, there is the problem that there is no uniform way how tools handle refactoring in more complex scenarios. These are some of the reasons why currently available tooling is not suitable for improving understanding of refactoring processes in an educational setting. We see here an opportunity.

2.3 Intelligent Tutoring Systems

The tools presented in the previous section focus on either detecting code smells, suggesting specific refactorings or performing automatic refactorings. These tools are not meant to teach students how to follow a proper refactoring process or how to avoid common mistakes. There does exist a type of tool designed to automatically teach skills; this type is the automatic tutoring tool.

Most intelligent tutoring systems (ITSs) have been developed within an educational context to assist students in accomplishing specific learning goals by providing them feedback as they solve problems. This feedback is often adapted to the student's own skills and context [2]. These systems provide a kind of personal teacher that can be available anytime, at any place. Tutoring systems have been developed in a wide variety of fields, but we have not been able to find an ITS focused on the software refactorings suggested by Fowler. Our search did, however, lead us to two recent studies that agree with our opinion that there is a need for research into including refactoring into education. In both studies, they suggest to research possibilities to include refactoring into an ITS [31, 63].

We have already explained that refactoring has the potential to improve code quality. By broadening our search to include tutorial tools that focus on code quality, we identified two tools that match this context.

FrenchPress is an ITS that targets intermediate SE students who already have a basic knowledge of programming. The software scans student code for mistakes common in object-oriented code, and it provides student-specific feedback to let students improve their code without a teacher. When the effect of this tool was measured, it showed positive outcomes, and it motivated students [14].

The second ITS we found was AutoStyle. The aim of this ITS is to improve coding style practices of students and help them to write more elegant code. An improvement in coding style is expected to bring an improvement in software quality. AutoStyle generates feedback by comparing submitted solutions for code exercises to model solutions. The feedback should help a student who submits a stylistically poor solution to develop a solution that is more elegant [39].

FrenchPress and AutoStyle also demonstrate two different approaches in generating feedback. The first provides hints without actively monitoring progress toward a correct result; the second actively monitors student progress toward a desired solution.

In both studies in which these tools are presented, the term refactoring is applied to the improvements students make to their code based on the generated feedback. This is accurate, while the code transformations suggested by the tools did not change the behavior of the code. From this perspective, we can say that we did find ITSs that focus on refactoring, but not specifically the refactorings suggested by Fowler. A major difference we can see here is that the ITSs we have found make, in many cases, code

improvements to single statements on local method scope level, while many of the refactorings presented by Fowler are more complex.

2.4 Relevance of Research

To summarize, our literature study shows that refactoring is a complex activity that needs a strict process. To perform refactoring in an optimal way, we need a good understanding of the process so that we can analyze the multiple options available and understand what effects each option will have on the end result.

We have seen that developers do not always follow a strict refactoring process, and that there is room for improvement in the understanding of the refactor process. Not following a strict refactoring process can have a negative impact on software quality after refactoring.

A good understanding of refactoring processes seems also relevant when relying on current automatic refactor tools. These tools seem, at this moment, not mature enough to allow users to fully depend on their outcomes. We addressed several issues that show we cannot blindly depend on these tools. The identified refactoring tooling is not suitable for the educational purposes we have in mind.

Some ITSs addressed the need for on including refactoring in their frameworks. Tools like FrenchPress and AutoStyle do generate feedback on refactoring code, but the type of refactorings they address are local changes on the method level that do not change more than a few lines of code. To the best of our knowledge, there are no ITSs that have incorporated particularly the refactorings suggested by Fowler.

From this summary we can conclude that that there is relevance and value in investigating the possibility of an educational tool that guides students in learning refactoring. With our research, we may provide a missing piece to help fill the knowledge gap that currently exists among studies of ITSs. When more knowledge and tooling are present in this domain, it might be interesting to see if tutoring systems aimed at refactoring actually lead to engineers who have a more thorough understanding of the subject and who can better analyze the effects of their refactorings, whether performed manually or automatically.

It is not feasible to develop in one study a complete educational tool for this purpose. As a first step, the next chapter describes an envisioned workflow that such a tool should offer. For each workflow step, we identify where existing work might be applicable and where still more investigation is needed. Based on this exploration, we identify the specific subject of our research and discuss the conditions of our research in more detail.

3 Refactoring Guidance Tool for Undergraduate SE Students

3.1 Introduction

This chapter presents our vision on a tool that could assist undergraduate software engineering (SE) students during their manual code refactoring activities. Based on the outcomes of our literature research, we first reason why existing automatic refactoring tools are not always suitable in an educational setting. Secondly, we look at the steps we can recognize when code is manually refactored and decide which specific process steps we like to include in our tool from an educational perspective. The outcome is formulated in a refactoring workflow that we think the envisioned tool should support in an automated way. We conclude this chapter by showing how our main research question is relevant and contributes in the context of the functionalities of our envisioned tool.

We mentioned in chapter 'Theoretical Background' some reasons why existing automatic refactoring tools are not optimal in an educational context. We add here another argument why these tools are less suitable: Existing automatic refactoring tools act as a black box. These tools do not demonstrate to a student which steps are necessary in a selected refactoring, why these steps are relevant and do not warn for possible side effects that the suggested changes might have on the software project. Previews of the suggested code transformations are offered by the tools, but how can we assume a student can judge these code previews as they only have worked with automatic tools and have no practical experience with refactoring?

There is to the best of our knowledge no specific studies available that have looked into how students perform refactorings in practice. Some hints are there that indicate limited experience from our observation of undergraduate bachelor software engineering students performing refactorings and the preliminary results from the end user evaluations performed in this study. The results suggests that students have difficulties in interpreting the results of automatic tools, cannot fully explain how the refactoring actually is performed and find it hard to foresee the consequences of their choices they have to make in automatic refactoring tools. In interviews we had with students (Chapter 6.3- Student Evaluation) a common answer was to "just press ok" when choices had to be made within the tools. To the best of our knowledge there are no studies to date that looked into issues students are facing when using automatic refactoring tools. However, we have found in our literature study, presented in chapter 2, that professionals who are using automatic refactoring tools face similar type of problems that we observed with our students [44, 60]. We assume that by teaching students on how to manually refactor code; this might give them more insights and skills in the refactoring process. This improved understanding might help them to better use existing tools and enable them to perform code refactorings without refactoring tools in those case were the tools do not provide a satisfying result.

Unfortunately, what literature also showed us is that code refactoring is not a skill quickly learned and professional acquire their skills and improve by working together with experienced peers [43]. This type of approach is what is known as apprenticeship learning, what also gave promising results in higher vocational education [7, 34, 62]. These studies inspired us in defining a vision where expert knowledge would be shared by a tool that offers code refactoring guidance to students as if an experienced peer was sitting next to them. Two questions we like to answer here: First, what should such a tool offer in an educational context? Second, which refactoring process should we follow that meets a professional standard?

We summarize here a few points that we think are important to include in our envisioned tool from an educational perspective:

- Expert knowledge on demand without the need of an experienced engineer or teacher.
- Tools presented in our literature work on fixed exercises [33, 63]. We think it is for a student more interesting if feedback could be generated for any piece of code a student is working on.
- Feedback that relates close to the code context, by e.g. using the naming of the code that is being refactored.
- Existing tools on static code analysis (SonarQube) offer hints for code improvement, but these hints are general and not specific to the task the students is working at. We think it might give more focus when providing hints for code improvement specific closely related to the refactoring that is being performed.
- Offering multiple solution paths for a refactoring and make it possible for a student to make choices in these paths, which adapt the solution to their needs as suggested by Abebe [1]. It also gives an opportunity to explore refactoring possibilities.
- Content of feedback can be adapted to the knowledge level of the student.
- Continuous looking over the shoulder of the student; presenting feedback when steps in the refactoring process are made that might lead to a wrong solution.

Next to the importance of these functional features we also want the tool to suggest a code refactoring process that resembles a professional approach. For this we have taken process steps suggested in a study that looked into what a professional and user-friendly refactor tool should offer [41] and augmented this with suggested steps in the refactoring process described by Fowler [19]. Finally, we looked at our functionalities that we defined from an educational perspective and determined if more steps are needed than the ones identified already based on literature.

We created a refactoring workflow from the process steps we consider relevant. The next section elaborates on this workflow.

3.2 Refactoring Workflow

We present here a workflow of which we think an ideal refactor guidance tool for students should implement the mentioned process steps. It would guide a student in the refactoring process from detecting code smells until determining whether or not the refactoring performed actually did improve the code. This workflow contains process steps which are depicted as squares (Figure 1). For each process step we define its goal and if the process step was originally suggested by Murphy-Hill (MH) [41]; Fowler (FO) [19] or was introduced based on the summary of educational requirements in the previous section (ER). We looked for each process step if the functionally potentially can be covered by existing research or if further investigation is needed. Our research will contribute mainly to step 4 – Risk Notification and step 5 – Refactoring instructions.

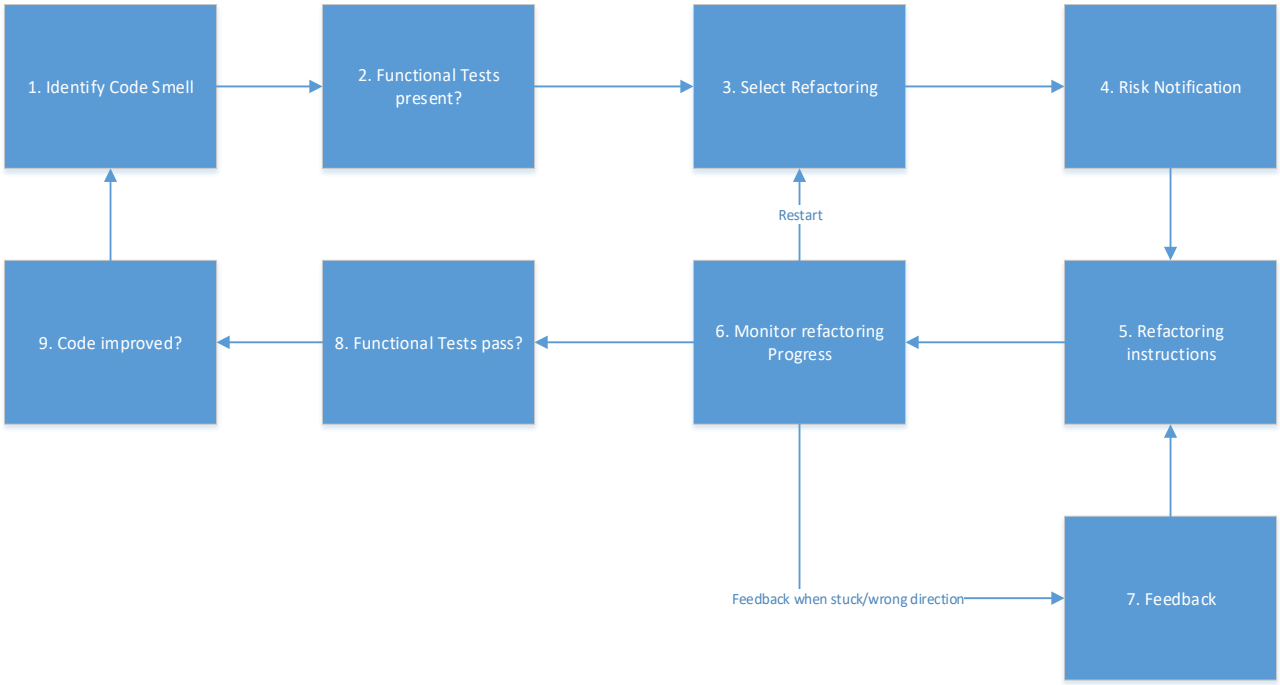


Figure 1 Flow between identified tasks in educational refactor tool

1. Identifying code smells (MH)
2. Functional Tests present? (FO)
3. Select a refactoring from a list of possible refactorings (MH)
4. Inform about potential risk when using selected refactoring **(ER)**
5. Instructions on how to manual refactor code (MH)
6. Monitor refactoring progress to a correct end result **(ER)**
7. Feedback when stuck or moving in the wrong direction **(ER)**
8. Functional Tests still Pass? (FO)
9. Did the code improve? **(ER)**

The original study of Murphy-Hill contained the steps “”review result” and “accept/reject”. These steps are omitted from the presented flow in Figure 1, because they are considered only relevant in an automatic refactoring case. Each of the steps depicted in Figure 1 is explained in more detail below.

Identifying code smells - The refactoring process starts by identifying code that is candidate for refactoring. These are typically code constructs which are suspected to cause software quality issues (*code smells*). The ideal refactoring guidance tool can present in this process step the code smells which have been detected and provide explanations why they are marked as such. Students can learn this way how to recognize code smells in code.

A large number of studies have looked into the automatic detection of code smells [20, 27, 34, 42, 51, 56]. The ideal refactoring guidance tool might integrate findings from this studies to detect code smells. Studies that also might be useful in this specific context can be the studies that have looked into the prioritizations of code smells based on the urgency to resolve [18, 61] or those that tried to figure out

how developers select code smells in practice [48]. Such studies might offer techniques to generate feedback that helps students in deciding which code smells are more relevant than others to resolve.

Check Functional Tests are in place - Any piece of code that we want to refactor should be guarded by sufficient functional tests in the project to assure correct functional behavior of the code after the refactoring has been performed.

Without any prior defined knowledge on the functionality present in a piece of code it is impossible to determine in an automated way if functional tests are missing. What we can do is determine if at least tests are present for the piece of code to be refactored. A simple overview of unit test code coverage might be an indicator of this. Some studies looked into the automatic generation of unit test cases before a refactoring is automatically performed [40, 53] by analyzing the code behavior for a range of input parameters. Maybe automatic suggestions for improvement of the present test suite can be given based on the techniques used in these studies.

Select refactoring - This process step could offer multiple solution paths for a selected code smell in the code as suggested by Abebe [1]. A solution path would be a specific refactoring strategy as those presented but not limited to by Fowler [19].

An example of a study that suggest an automatic refactoring based on a code smells is JDeodorant [58]. A single solution path is determined for code smells detected in the code

Risk Notification – Students are informed in this process step about constructions in the code being refactored or code that depends on the refactored code that need extra attention. This pieces of code might poses a risk. Students are informed to analyze some pieces of code for possible side effects in their specific code context. We identified changes in code that have:

- Direct impact
  Code that is changed because of a certain refactoring might possibly lead to a change in behavior. For example: When performing a rename of a method that is part of an override relationship can have a direct influence on the behavior of the code when other code depending on this method is not analyzed properly.
  A choice could be made to automatically analyze for problematic dependencies, but we could also decide to let students investigate the problem and make an own decision.
- Indirect impact
  Code that is changed during a refactoring that does not influence behavior in the existing code base, but might have an impact on future code baselines or external packages. For example: When performing an Extract Method refactoring and making the new method *public* will extend the public interface or class. Future users of this class might make use of this new method, which can result in unwanted side effects.
  A possible suggestion that could be presented to students is to make new methods *private* by default and only make them to public if the design or compiler is requiring this.

We see similar indirect impact risk identification in static code analysis tools like e.g. SonarQube and PMD. The disadvantage of these tools is that the number of presented warnings can be large and are not particularly focused on the refactoring activity a student is working on, which might make it hard for a student to put presented solutions in the right context. Automatic refactor tools like IntelliJ and Eclipse

might take in most situations the direct impact scenarios into account. However, this is done in a black box manner. This approach does not give students information which risks were taken into account and how the tool anticipated upon these risks.

We added the process step "*risk notification"* because we think that in an educational perspective it makes sense to issue warnings that might improve code that are specific to the refactoring a student is working at. Second, one of our goals was to give students insight in the refactoring process so students should be presented the direct impact results, which they can analyze themselves.

To the best of our best knowledge there is at this moment no studies that combined risk analysis of code together with instructions on how to refactor a piece of code in an educational context.

Refactoring instructions – This process step generates instructions on how to perform manually a refactoring step by step. The instructions should be generated clear and related closely to the code for which the manual refactoring instructions is generated. This could be done by including names that are used in the code and adjust the instructions based on context of code.

An example on how instruction steps might change in different code contexts is presented in the two examples blow.

```
protected void MethodExtractA(){
    int a = 0;

    a++;         // Extract this line to new method

    System.out.println(a);
}
```

The first example shows a code fragment that resides in a code context where variable *a* is instantiated locally and used only there. In this specific case, manual refactoring instructions could tell to move the marked code to a newly created method that returns the updated value of *a* to the original calling method. In this method the value of *a* is updated and used later on in *println*.

In the second example below we change the code context. *MethodExtractA* is now declared to have *a* as an input parameter and to return the value of *a*:

```
protected int MethodExtractA(int a){

    a++;         // Extract this line to new method

    System.out.println(a);

    return a;
}
```

In this case the generated instructions from the first example still would result in a proper extract method refactoring, but it does not have to be the best solution in this case. Deeper analysis of depending code that uses *MethodExtractA* might reveal that the value of a is used in other methods that also return this value back to their callers. It behaves as a carrier of data through the methods of the same class. In this case a better choice might be to transform local variable *a* to a member variable of the containing class and adjust also the other internal methods to which the value of a is passed on.

We want students to be aware of the influence of code context and offer them at least choices to investigate before starting the refactoring. We think that generating instructions that is based on this deeper analysis of code that extends beyond the local scope of a method can prove very useful for this purpose. The intelligent tutoring systems we have identified in our literature study all address their feedback on only the local scope of a method.

The possibility of generating instructions on how to manually perform refactorings described by Fowler and where code context analysis goes beyond the local scope of methods is something we have not seen in any other tool studies.

Monitor Refactoring Progress – This step would evaluate how the student makes progress to a possible end solution of his refactoring. This could be non-stop, by request or something that is a mix of both.

There are studies available that determine how well the code matches one or more model solutions [21]. These presented solutions work often on code exercises of which the expected outcomes are known in advance. In our specific case, were we want to be able to use the tool for any piece of compilable Java code, we might consider generating models based on the outcomes of the "refactoring instructions" process step. We also look into this possibility later in chapter 7.2- Future work.

Feedback – Based on the progress of a student relevant feedback should be provided on how to proceed or how well the progress is. The same research as in the previous steps looks into this question.

Functional Test pass? - To assure that the code changes made during the refactoring did not alter behavior the set of tests which were defined in step 2 can be executed and results shown.

Besides that tests that compile do not pass anymore we can distinguish two other cases that also should be addressed:

1. Non compilable test code as a side effect of the refactoring, e.g. missing methods because of a 'rename method' refactoring.
2. New introduced code for which no test code exists, e.g. new added classes because of a 'move method' refactoring.

For the first point addressed it might interesting to include detection of breaking test code in the step 'monitor refactoring progress'. For the second point addressed a post analysis of code coverage and new added methods might be considered in the step were we evaluate if functional tests still pass.

An interesting study that contributes a solution for our mentioned cases [47] might be interesting to considered in our ideal refactoring guidance tool.

Code improved? – Concluding the workflow we would like to offer the student insight if the code quality indeed improved. The provided information can be used by a student to decide to revert the changes and follow a different refactoring strategy or accept the resulting code.

Several studies have been looking into how we could determine code quality in relation to refactoring [3, 50]. Another study looks into how code metrics can be used to estimate the impact of a refactoring [11]. We can in the last case a possibility to track the particular metrics related to a refactoring and determine here if they actually improved or not.

After this process step we can continue again to step 1 that will start identifying code smells in the new code base.

3.3 Requirements and Scope of Research

We showed that there is for many of the process steps in our presented workflow a body of knowledge available that could be used as a starting point for future work.

The process step that will have our main focus is step 5: “Refactoring Instructions”. To the best of our knowledge there has been no specific research done in this specific area and we have shown earlier in our literature study, that studies in the domain of intelligent tutoring systems also mention the topic of refactoring as an issue to further investigate. Step 4: “Risk notification”, is closely related to the generation of refactoring instructions; we will also pay attention to this step in our research.

3.3.1 Scope

The scope of our project will be:

- End-user group: Undergraduate Software Engineering (SE) students.
- Prototype can analyze: Compilable Java code, standard edition, version 8.
- Fowler’s Refactorings: *Rename Method* and *Extract Method*.

3.3.2 Functional

In our research we will focus on the automatic generation of refactoring guidance which is based on code context. This maps to process step “refactoring instruction” in Figure 1. *Refactoring guidance* is an overview of instructions on how to manually perform a selected refactoring upon a piece of code in which code context is taken into account. The code context will be used to generate refactoring guidance that is relevant to the code context in which the refactoring takes places. The exact definition of terms and how the instantiation exactly is done is part of our research and will be explained in chapter 5 in more detail.

Based on the analysis we presented in this chapter, the following functional aspects will be taken into account when generating refactoring guidance:

- Automatic generation of refactoring guidance based on code context is possible for any given compilable Java project written in Java 8 SE.
- Generated refactoring guidance contains instructions that are closely related to the code provided, e.g. uses the namings of variables retrieved from the code context.
- When possible multiple solution paths are presented out of which a student can choose which paths to explore and/or follow.
- Hints are included to warn the student for possible side-effects that the refactoring might have.

In this study a prototype is developed to evaluate the automatic generation of refactor guidance, which will meet the above functional requirements.

3.3.3 Non-Functional Requirements

The prototype implements 2 out of 72 possible refactorings mentioned by Fowler in his book. For this reason, a non-functional requirement is added to make extension of the software prototype with future refactorings as easy as possible. The software design of our prototype will take this non-functional requirement into account.

The software prototype is a standalone tool in which a user can generate refactoring guidance based on a selected refactoring that is applied upon a code fragment selected in pre-defined software examples or in a provided software project.

3.3.4 Selected Refactorings

Refactorings are performed to solve code smells present in code. We looked for studies that analyzed common code smells introduced by undergraduate SE students in order to determine which refactorings are most relevant to address into in our study. Some studies look into code quality aspects, but none specifically into the code smells as mentioned by Fowler. We decided for this reason to do a quick-scan of Java projects that were the result of work by third year undergraduate students. From this quick-scan we concluded that it would be most interesting to aim our research on the instantiation of refactoring guidance for *rename method* and *extract method*. This outcome is in line with another study that looked into most common refactorings of professionals. Details describing the set-up and outcome of our quick scan can be found in *Appendix K – Code Smells Quick-Scan of Java projects* .

4 Research Approach

4.1 Context

In the previous chapters we explained why it is relevant to develop a refactoring guidance tool that assist undergraduate SE students to learn and understand how to refactor code. We presented a vision that describes the desired workflow offered by such a tool.

The specific topics we want to address in our study, that could fill in a knowledge gap in the desired workflow, is on the automatic generation of refactoring guidance from an arbitrary piece of code that could be used to guide students in their learnings on how to refactor code. Specifically, the refactorings as described by Fowler. The generation of refactoring guidance is based on code context. The specific refactorings to look at are *rename method* and *extract method* as is explained in the previous chapter.

This chapter describes how we setup our research to come up with a possible answer to the above.

4.2 Research Questions

The main research question to be answered in our study is:

> *"How can we automatically generate refactoring guidance for Java code that is based on code context? "*

We formulated two sub research questions to answer this question:

I. How should we, according to expert knowledge, perform refactorings *rename method* and *extract method* manually?
II. How can we generate instructions on how to manually refactor Java code, that are adapted to code context?

Sub-question I. should result in an answer that provides an exploratory overview of what expert knowledge says about performing the refactorings *rename method* and *extract method*. The knowledge acquired by this question is used to define the content of the automatically generated refactoring guidance. It should also provide us with insight on which specific code context can have an effect on the content of the generated refactoring guidance. Because of the exploratory nature of our study, the intend is not to acquire a fully complete knowledge on all cases for the mentioned refactorings.

Sub-question II. should result in a theoretical model and algorithm that can be used to generate code context based refactoring guidance for a selected refactoring of a piece of code that is part of a compilable Java 8 SE project. The model should be able to hold the expert knowledge that has been identified in sub-question I to make sure that our guidance is based on expert knowledge.

4.3 Iterations

The study has been executed in three iterations, where each iteration addressed one specific focus point. We will elaborate on this below.

The first iteration provided a partial answer to sub-question I: How should we, according to expert knowledge, perform a *rename method* and *extract method* refactoring? We answered the question by literature research and evaluations of the literature research outcomes with peers. Within this iteration we focused on the *rename method* refactoring. Together with the expert knowledge we identified which properties of code context can have influence on how a refactoring should be manually performed.

The second iteration resulted in the theoretical model and algorithm that is able to hold the expert knowledge identified in iteration 1. A proof-of-concept in the form of a software prototype has been developed to evaluate the model and algorithm. This first iteration was focused on generating code context based refactoring for *rename method*.

In the third iteration our prototype has been extended for *Extract Method*. Additionally, important outcomes of user evaluations in iteration 2 have been included in the prototype.

4.4 Verification and Evaluation

A proof-of-concept in the form of a software prototype has been built that is used to verify the correct behavior of the theoretical model in a practical setting. To evaluation of the model possible, concrete models that contain expert knowledge have been constructed for *rename method* and *extract method*.

To evaluate the generated refactoring guidance based on our presented model and algorithm we ran defined use cases for *rename method* and *extract method*. For each use case a specific Java code sample was created of which we knew how the output of the generated refactoring guidance should look like. These samples were given as input to our prototype. The generated refactoring guidance by the prototype was compared with the expected result.

To verify expected behavior of our prototype on a smaller granular level like instantiation of instructions in isolated code contexts and implemented algorithms needed for code analysis were verified by a large set of unit tests.

To have early student involvement and feedback we ran end-user evaluations after iteration two and three. We have chosen after iteration two for individual semi-structured interviews. In iteration three, two semi-structure group interviews were taken with third year undergraduate SE students. The main purpose of these evaluations was to: gain insight if students consider the prototype and concept to be useful in an educational setting, get an insight how students refactor, collect feedback for tool improvement and gather suggestions for future functionality.

5 Generation of Refactoring Guidance based on Code Context

This chapter details how refactoring guidance can be generated based on code context from both a theoretical and practical perspective. The first part of this chapter provides a theoretical answer to Sub-question II:

> How can we generate instructions on how to manually refactor Java code, that are adapted to code context?

First, the main concepts of our theory are introduced. Second, we introduce a model that is subsequently used to generate code contextual refactoring guidance. In Section 5.3, an example is provided to explain the algorithm we use to automatically generate instructions based on a piece of code, its code context and our introduced model.

During this study, a prototype is developed that is used to evaluate the presented theory in practice. Section 5.4 elaborates on some aspects of this prototype.

As will be explained, the basis of our theory lies in generating refactoring guidance based on code context. We conclude this chapter by explaining which process we followed to identify for a refactoring: how it should be performed according to expert knowledge and which specific code context can influence the content of the generated instructions. We work this process out in more detail for the refactoring *rename method*. The same process we used to analyze the refactoring *extract method*. An overview of these results is given in Appendix B – *Identified Extract Method* . The analyses of this specific refactorings provide us with a partial answer to Sub-question I:

> How should we, according to expert knowledge, perform refactorings rename method and extract method manually?

5.1 Main Concepts and Terms

The main concepts and terms are detailed here to ensure a good understanding for the subsequent presentation of our theory.

5.1.1 Code Refactoring

Code refactoring = "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" (Fowler, 2000).

Fowler describes how these changes to the code should be actually executed for 72 different code refactorings. In his definition of *code refactoring*, the changes can be either performed manually or automatically.

---

refactoring activity = the *manual* changes made to a piece of code to achieve a specific code refactoring.

In the context of our work, the idea is for a student to make manual stepwise changes to the code by following the process for a specific code refactoring, as described by Fowler. We introduce the term *refactoring activity* to distinguish between *code refactoring* and the manual activity of changing the code

5.1.2 Refactoring Subject

---

refactoring subject = the code fragment that is cause for the programmer to execute a specific refactoring.

Figure 2 shows a code fragment where a student decides that a rename method code refactoring is necessary on *readData()* in the class *Example*. The location of *readData()* in the code should also be the starting point for the student to analyze what impact the renaming of this method would have on other parts of the code and, second, if any other additional changes are needed in different places in the software project. In this specific case, there are probably also changes needed in the class *Base* and in the interface *IExtern*. The code fragment that is the cause for starting a refactoring is called the *refactoring subject* in this research. In this example, the refactoring subject is *readData()* in class *Example*.

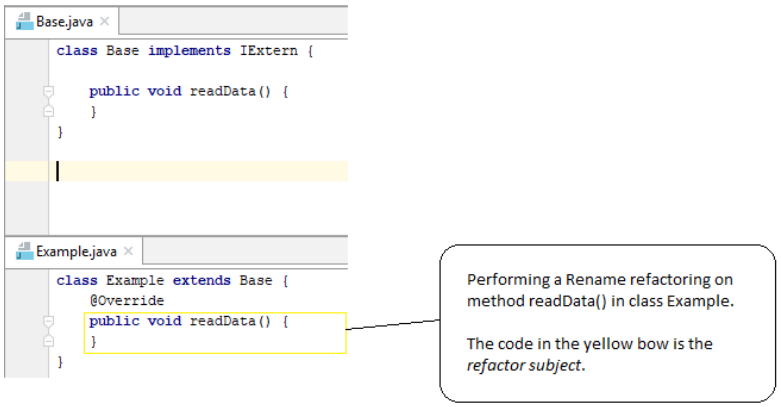


Figure 2. An example of a refactoring subject

5.1.3 Refactoring Guidance

***Refactoring guidance** = the instructions on how to manually perform a refactoring X upon a piece of code, in which code context is taken into account. The guidance also includes instructions as to which parts of the code might need special attention during the refactoring activity and why certain instructions are needed.*

From our literature research, we learned that an experienced software engineer should execute his refactoring activities based on the knowledge he or she has gained from peers or from past refactorings. The engineer knows what the possible variations are for a specific refactoring, what are the attention points in the code context, and what the pitfalls might be.

A novice student who applies a specific code refactoring for the first time might have difficulties with this activity, because of a lack of experience in the task. Apart from a lack of knowledge about code refactoring, it is also sometimes hard to directly see which code fragments in a code context have influence on the way in which a refactoring activity must be executed. These code fragments can be located far from the refactoring subject itself. An example of this is given in section 5.1.4.

Chapter 3 explained our vision of a refactoring guidance tool that helps students to learn code refactoring, as if an experienced engineer were giving feedback to them on the steps to be taken. We also explained in this chapter that our study focus is on the automatic generation of refactoring guidance that is based on code context.

Refactoring guidance is constructed from pieces of textual instructions that are signified by specific code context properties. Code context and its properties are explained in Section 5.2. The textual instruction is called an *advice* in this research. For the term *advice*, it must be emphasized that we consider the instructions as guidelines and that they are open to alternative approaches by students. In this regard, our goal is to provide insights into the refactoring process and to the code context where a student can explore possibilities; it is not the intention of this work to lead students to one fixed solution.

In our study, we distinguish between the following three types of advice:

- General instructions
- Recommendations
- Warnings

General instructions are instructions on how to perform a refactoring activity where there is no need for additional information. These are often the core instructions of a refactoring such as, for example, within the rename method, where there is always instructions present in the guidance that tells to recompile the code, and solve any unresolved method compiler errors.

Recommendations are instructions for how to improve readability, maintainability, or understandability of the code; it should be noted, however, that these manual steps are not necessarily needed to maintain the correct behavior of the code. An example of this in the rename method refactoring procedure is the recommendation on renaming those methods that are overloaded (*see* the code example below). Renaming only one of the “print” statements will not result in a change of behavior, but for understandability of the code it makes sense to recommend to rename also the second “print” method. This type of advice should also include information as to why these steps improve the readability and/or maintainability of the code.

```
public class A
{
   public void print(String str);

   public void print(int value);
}
```

Warnings are instructions that address a particular situation in the code context that might possibly lead to problems when a refactoring is manually performed without taking the suggested code context into account. A student should investigate if the presented risks are actually relevant and should decide whether or not to execute the advice. This type of advice gives the student deeper insights into the structure of the code. An example of a warning, while renaming a method, is the instruction to leave the

original method name *@deprecated* in the interface definition, because the method being renamed is publicly exposed via a public interface. A student should investigate, in this case, if any external packages depend on these interfaces and should ascertain whether or not the instruction for making the method deprecated is really needed. A piece of advice can now be defined as follows:

Advice = General instruction *or* Recommendation *or* Warning.

Refactoring guidance can be defined as follows:

Refactoring guidance = Consists of one or more *Advices.*

5.1.4 Code Context

In the existing literature, the term *code context* is used often, but an exact definition is lacking. We think it is useful at this point to clarify the term in our study. A general definition for context is given by the Cambridge Dictionary [10]: "the situation within which something exists or happens, and that can help explain it."

Starting from this definition, we can precise what code context means in our study. The activity in our research is the *refactoring activity*, to be applied upon a refactoring subject. The refactoring subject is not an isolated piece of code but is part of a software project. Within this project, the refactoring subject can have all kinds of relations with the surrounding code in the project such as, for example, class and interface relations, calling/caller relations, and type-binding relations. These relations create a situation for the refactoring subject that might influence how to perform a specific refactoring.

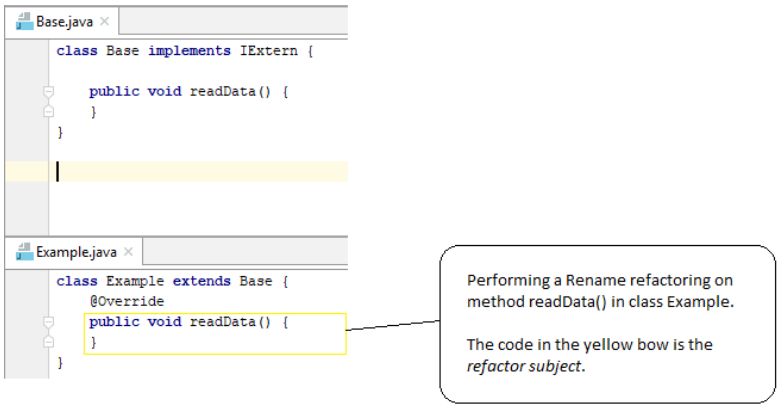


Figure 3. Performing the rename method on the refactoring subject

A concrete example of these influences on a *refactoring subject* can be seen in Figure 3. The refactoring subject is method *readData()* in class *Example.* This class has been extended from the class *Base*, which implements methods described by the interface *IExtern*. In both the interface and the class level, the same method name is present as being used in the refactoring subject . All the code fragments that are involved in these relations are considered in this research to be part of the code context of our

refactoring subject. Next to these external relations, the refactoring subject itself is also considered part of the code context. For example, we create an advice for renaming the refactoring subject and want to include the original name of the method in this advice. In order to do this, we retrieve information out of the code context, which, in this example, entailed the method name before it was renamed.

---

code context = the refactoring subject and all other codes that have any direct or indirect relations to the refactoring subject.

When renaming method *readData()* of the refactoring subject in Figure 3, we had to at least consider in our refactoring activity, first, what to do with the override relation in the example and, second, the fact that the base class implements an interface method that dictates the method name of our refactoring subject because of the override relation present. If the method name of the refactoring subject had been only declared for the first time in the class *Example*, then the other classes would not have been considered part of our code context because they would have had no direct or indirect relationship to the refactoring subject. This example also demonstrates how *refactoring guidance* varies in different code contexts.

The example in Figure 3 demonstrates also from how far the code context can influence the generated refactoring guidance on how to refactor the refactoring subject. A public class or interface with several layers in the inheritance hierarchy, and where code from external packages use the public interface, will trigger necessary renaming instructions in many classes and packages if we rename a method in the bottom of the class hierarchy. It might be hard to oversee all the needed changes for an unexperienced student when there is no guidance present.

Within our study, we have focused on the analysis of the code context that stays within the boundaries of one Java software project; thus, external projects depending on the project of the refactoring subject are not integrated into our code context analysis. We do, however, provide advices to inform about the presence of risk when external projects might be involved. For example, the refactoring guidance includes an advice if the method in the refactoring subject is publicly exposed through a public interface. In these cases, it is left to the student to investigate if the mentioned risk is relevant. As stated earlier, our main goal is to provide insights to students rather than complete refactoring solutions.

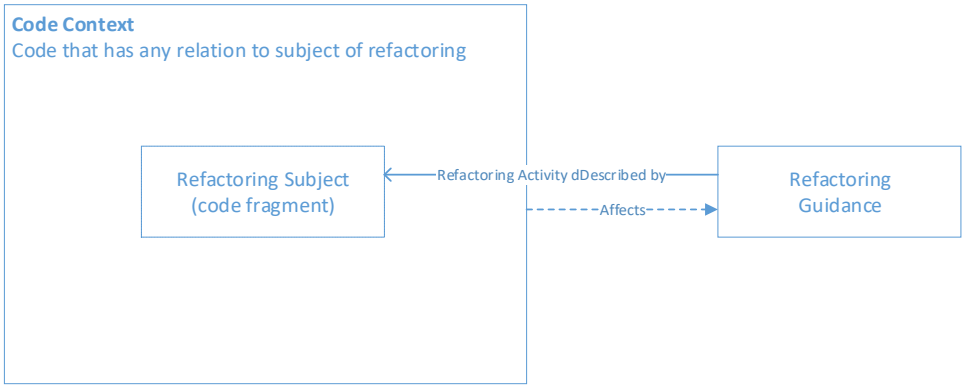


Figure 4. A visual representation of the code context of our refactoring subject, showing how code context can influence the instructions that describe how to manually refactor code. The dashed lines around the refactoring subject indicate that the refactoring subject itself is part of the code context.

Figure 4 depicts visually our presented definition of code context. It also shows that refactoring guidance describes the refactoring activity and how its content can be changed by the code context.

5.2 Model

In this section, we introduce a model that contains the necessary information from which we can generate *refactoring guidance* that is based upon *code context*. The first thing we examine is how to create *advices* based on the refactoring subject's code context.

5.2.1 Code Context Property

---

code context property = a code construct present in the code context that can be associated with one advice template.

We showed earlier, in Section 5.1.3, that code context can influence refactoring guidance. When analyzing the rename method and the extract method, we discovered that we can associate advices with specific code constructs that are present in the code context. We call such a specific code construct present in the code context a *code context property* (CCP). The code context properties we found for the *rename method* and the *extract method* are presented in Chapter 6—*Evaluation*.

An example of CCPs can be explained by taking the sample code in Figure 5 and consider the scenario when renaming method *readData()* in class *Example*. The code context in this example is the method being renamed and all depicted classes. We can, among other, identify two CCPs in this code example: the method *readData()* override by super class method (CCP_1), and the method *readData*() declared in a public interface (CCP_2).

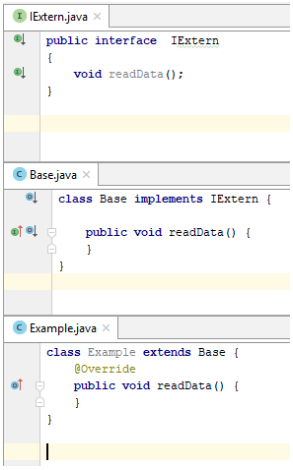


Figure 5. Example of the rename method refactoring for method readData()

CCP_1 can be associate with an advice of type *warning* that notifies the student that, when renaming *readData()* in class *Example,* one should consider also the overridden method in class *Base*. Only renaming the method in class *Example* might result in functional side effects ($Advice_A$). CCP_2 can be associated with the advice of type *warning* that informs the student the method being renamed has been declared in interface *IExtern*. It is recommended to investigate if the public exposure of this method is being used in software packages that might not be part of the project ($Advice_B$). These advices can be generalized to any method being renamed of which its code context holds these specific CCPs.

The override advice ($Advice_A$) might resemble the text depicted in Figure 6. The keywords, denoted by the hashtag symbol (#), are used in this advice because we want to define an advice template from which we can instantiate concrete advices that contain names of variables that are present in the code context. For this reason, a CCP is associated with an *advice template* (Figure 6) in our theory. How an advice template is instantiated, is explained in section 5.2.2.

```
Method #method has been defined in the
following superclasses: #class-list

To eliminate any side-effect risks, I
suggest to rename #method also to your new
name in class: #class-name
```

Figure 6. Advice template for "method overridden"

From this example, we can now conclude that when performing a rename method that the presence of CCP_1 and CCP_2 in the code context of the refactoring subject (in this case, the method being renamed), leads to template advices: $Advice_A$ and $Advice_B$.

By identifying all possible CCPs, relevant for a refactoring, and their associated advices, provides the information needed to create specific advices matching a given code contexts. An example of a refactoring with its advices and associated CCPs has been worked out for *rename method* in section 5.5.

Refactoring guidance that is built from these created advices is, thus, also automatically code-context dependent because the guidance is constructed from advices (See definition refactoring guidance in section 5.1.3) that are associated to the CCPs of the refactoring subject.

5.2.2 Code Contextual Advice Function

In this section, we explain how a CCP in the code context of the refactoring subject is leading to concrete advice.

A CCP is detected by a *code context property detector* (CCPD) function. This function evaluates the code context of the refactoring subject for the presence of a specific CCP. There is a CCPD function for each CCP that has been identified for a specific refactoring. In case of the rename method refactoring we have six CCPDs, as is shown later on in section 5.5. The CCPD function takes as input parameters code context and the refactoring subject. When the specific CCP is detected in the code context of the refactoring subject, the result of the CCPD function is an advice template that is associated with the detected CCP (Figure 7) as has been explained in section 5.2.1.

Figure 7. The CCPD function visualized

To create concrete advice, which is used later on in the refactoring guidance, we take the resulting *advice template* and determine, based on the *code context* of the refactoring subject, what the values of the hash-tagged keywords should be. This translation of hash-tagged keywords to concrete values is done by an instantiator Function I, which is depicted in Figure 8.

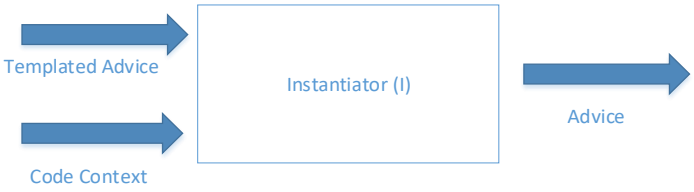


Figure 8. The Function I visualized

The instantiator Function I is generic: it will always instantiate *advice* from an *advice template* by filling in the hash-tagged keywords based on a given code context. Having defined the CCPD and the instantiator function, we can now merge them to form the generic code contextual advice (CCA) function (Figure 9). The CCPD function is variable in this CCA function, because we have a list of CCPD functions: one CCPD function for each identified CCP relevant for a specific refactoring. The CCA function takes as input parameters: code context, the refactoring subject, and a CCPD function. The CCA function has as output: a concrete advice.

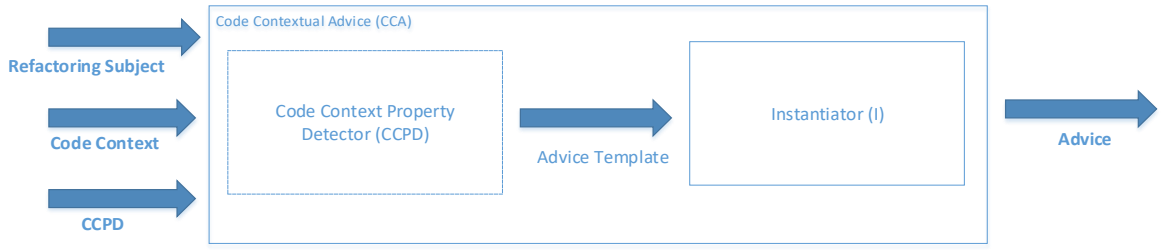


Figure 9. The CCA function visualized

5.2.3 Refactoring Advice Graph (RAG)

In the previous section, we demonstrated how to create advices for a specific refactoring X by evaluating with a CCA function a list of CCPD functions, one by one, together with a given refactoring subject and its code context. The result is a list of concrete advices for each CCP present in the given code context. Advices are created in order of the evaluation order of CCPDs by the CCA function. The presented theory does not have means to control the evaluation order. There are at least two reasons why controlling the evaluation order of CCPDs is important:

- Increasing the understandability and usefulness of the generated refactoring guidance.
- Preventing conflicting advices in the generated refactoring guidance.

Increased understandability and usefulness. There are situations thinkable in which advices should appear in a specific order in the refactoring guidance to make the guidance more understandable or more useful. For example, for extracting a method, the refactoring guidance might be more understandable when first giving the advice on the parameters to be parsed to an extracted method and, after that, the advice on how and which values to return.

Preventing conflicting advices. The analysis of the *extract method* refactoring showed us that the simultaneous presence of some particular CCPs in the code context of the refactoring subject, might result in a set of conflicting advices. An example of this can be given by taking two CCPs that can be present in the extract method refactoring. CCP_1: Extracted code should return one value. CCP_2: Extracted code contains a conditional return statement. The generation of a list of advices in any order goes well when only one of the two CCPs is present in the code context of the refactoring subject. However, when both CCPs are present in the code context this would result in two advices that are conflicting because they cannot be followed up simultaneously:

1. To return one value containing true or false for the conditional return.
2. To return one value for the parameter changed in the extracted code and this parameter is used later on in the original method.

A solution to prevent conflicting advices is *to order* the evaluation of CCPD functions. This ordering helps us to anticipate on situations like in the example case. We will see later on in chapter 6.1.1.2 how this ordering helped to solve the issue presented in the example.

The evaluation order of CCPD functions have been done by storing all CCA functions and their unique CCPD function parameter in a graph, which we call a refactoring advice graph (RAG). An additional advantage of this graph is that we can link vertices to any other path, which makes the reuse of advice possible. For example, when renaming a method, there is at some point a general advice that explains to

manually rename the method, recompile and solve the unresolved method compiler errors. From every path in the graph, we can finally go back to one definition of this standard advice for renaming the method. Figure 10 depicts this by the CCA "default" edge, by which we reuse "$Advice_d$". We introduced this special CCA function to enable the possibility to unconditionally go to a next advice vertex for those advices that are common and are given unconditionally.

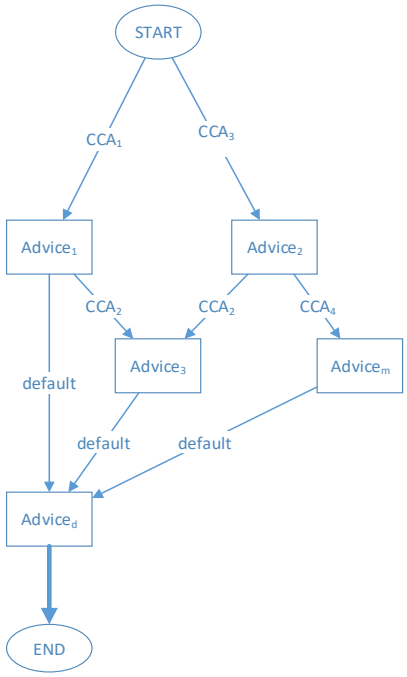


Figure 10. A schematic overview of a refactor advice graph (RAG)

Other important aspects of Figure 10 are explained here further. Each RAG contains a start vertex from which one or more edges leave. Each RAG ends with an end vertex. Each vertex in the RAG is a created advice that is the result of the CCA function on the incoming edge. Each edge in the RAG contains a CCA function that takes as a fixed parameter one of the CCPD functions that are defined for a refactoring X. Each CCA function that has the same CCPD function as a parameter must lead to the same vertex. This is in order to ensure the condition introduced earlier that each CCP should be associated with one unique advice template. An example of this restriction is given by the vertices "Advice1" and "$Advice_2$". From both vertices, we can see edges leaving with the same CCA_2. These edges both lead to "$Advice_3$".

We added two more restrictions alongside the already-mentioned restrictions in the RAG. These additional restrictions are needed to ensure that an algorithm can always traverse the graph from "start" until "end" vertex. The following restrictions makes this possible:

a. The RAG should contain no loops. Loops might lead to situations where we can never reach the end vertex of a graph. This means, in practice, that we cannot generate a refactor guidance.
b. From each vertex in the RAG, there is in a given code context always exactly one edge of which the CCA function results in a concrete advice. Without this restriction, two problems could be encountered.

1. We might, again, not be able to reach the end vertex of the graph because when none of the CCA functions on the leaving edges of the vertex lead to an advice we cannot proceed further in the graph.
2. Allowing more than one CCA function on the leaving edges of the vertex that leads to an advice introduces non-deterministic behavior. In this case it is not clear which advice should be included in the refactoring guidance.

Restriction (b) means that the CCPDs that are evaluated by the CCA-function on the leaving edges of a vertex cannot be treated as isolated cases. We must ensure that the CCPDs being evaluated on the outgoing edges of a vertex are mutually exclusive and that always one of the CCPDs will be true in any situation. We call the group of CCPs that are evaluated by these specific type of CCPDs a *code context property group*.

A RAG, as explained here, is defined for each specific refactoring. In our study we have constructed initial RAGs for *rename method* and *extract method* as presented later in chapter 6.

5.3 Algorithm

Before explaining how our algorithm works, we define the artifacts that are needed as input to generate refactoring guidance. Figure 11 gives a schematic overview. The algorithm needs to know what the refactoring subject is and where it is located in the provided context. Typically, the location is defined by a filename and the code lines where the refactoring subject can be found. We also provide the code context of our refactoring subject. The code context can be a single class (for example, extract method refactoring), one Java project (for example, the rename method), or even multiple Java projects when we want to take into account dependencies with external packages in case of, for example, the rename method. The last artifact needed is the RAG that belongs to the refactoring that we want to perform. Based on this input, the algorithm can generate refactoring guidance based on code context.

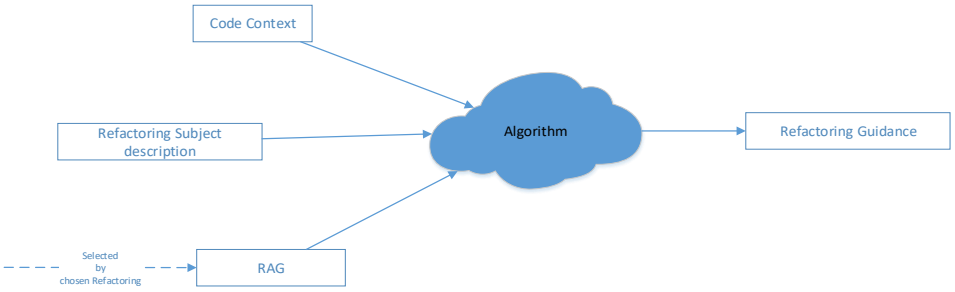


Figure 11. The input and output of our algorithm to generate refactoring guidance based on code context

5.3.1 Execution Sequence

The steps that our algorithm takes can be described schematically by means of an execution sequence for how to generate refactoring guidance (see Figure 12).

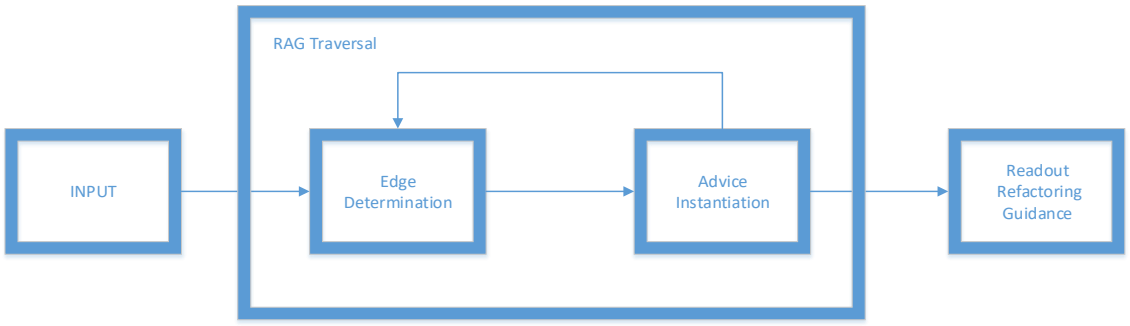


Figure 12. Refactoring guidance generation—execution sequence

We used the example introduced earlier where a student wants to rename the method *readData()* in the class *Example*. Figure 13 illustrates the example and shows some relations from the RAG to the example code that are explained later in the 'RAG traversal' stage of the execution sequence.

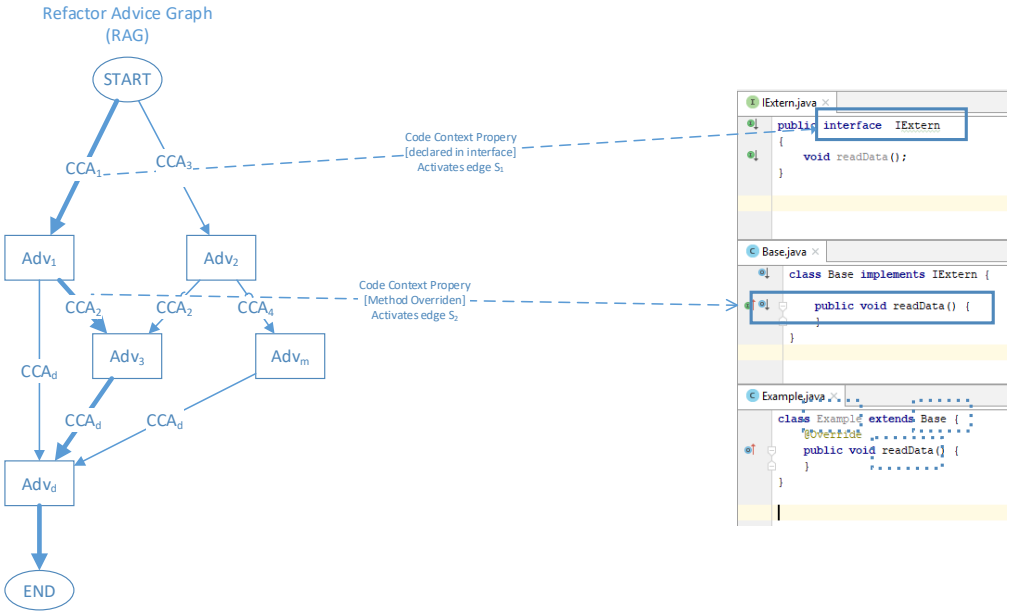


Figure 13. RAG and code example

Input

In the input stage, we provide the algorithm with the necessary input artifacts. In this case, a student would indicate that he or she wants to perform a rename refactoring. This choice will select the RAG that holds information for this refactoring. The student would also indicate the Java project (*code context*) that contains the files of our example as well as in which class and for which method the renaming will take place (*refactoring subject description*). This could be done by providing the filename and the line number or, in a more interactive way, by selecting the method directly in the source code. When this information is provided, the algorithm can move onto the next stage.

RAG Traversal—Edge Determination

After the input stage, we move to the RAG traversal stage, in which a path should be determined that is used later on to instantiate the output for the refactoring guidance. This stage is characterized by a cycle of two internal stages that continues until the “end” vertex of the RAG is reached.

Based on the input of the “input” stage, the corresponding RAG is loaded, and the algorithm starts at the “start” vertex. The algorithm will evaluate the CCA functions that are defined on the outgoing edges. The algorithm will provide the CCA functions with the code context , refactoring subject and the CCPD function which is defined for each edge in the RAG. In our example (Figure 13) we can see that from the start vertex there are two edges available that evaluate CCA functions “CCA_1“ and “CCA_3“. “CCA_1” contains a CCPD for detecting the code context property ‘method defined in interface’. “CCA_3” contains a CCPD for detecting the CCP that the method of refactoring subject is *not* declared in an interface. In this case, the method of the refactoring subject has been defined in an interface; thus, the CCA_1 function will result in a concrete advice.

RAG Traversal—Advice Instantiation

The edges where the CCA functions that lead to a concrete advices in “edge determination” are followed. The concrete advice that was instantiated during “edge determination” is stored in the vertex that has this specific incoming edge. In our example “Adv_1“ will contain the concrete advice. After this step, the RAG traversal continues again with an “edge determination” that is based on the outgoing edges of the vertex “Adv_1“. In our example this means that from vertex “Adv_1“ function “CCA_2“ leads to vertex “Adv_3“. This “edge determination” and “advice instantiation” continues till the “end” vertex has been reached.

Readout Refactoring Guidance

When the “end” vertex has been reached, we have defined a path in the RAG that contains advices that are specific for the code context of the refactoring subject. The algorithm will concatenate, in the last stage, the contents of all the vertices on the found path to a refactoring guidance text. This could be depicted as follows in our example:

$$\textit{Refactoring guidance for renaming the method of the refactoring subject} =$$

$$\textit{Advice}_1 + \textit{Advice}_2 + \textit{Advice}_d$$

This refactoring guidance can now be presented to the student.

5.4 Implementation

The model and the algorithm presented in this chapter are evaluated by implementing them in a software prototype. Some important technical and design decisions of this prototype are explained in this section in order to help the reader understand how the presented theory relates to the practical implementation of our prototype. The software design of our prototype is presented in Appendix D – *Prototype* Design.

The prototype’s source code is publicly available at: https://github.com/patrickdb/RefactorGuidance

5.4.1 Code Context Property Detection

In our design, the responsibility to detect one specific code context property, that is used in the CCPD functions, is implemented by those classes that inherit from type *ContextDetector*. These specific classes can be seen as the implementation of the specific *code context property detector function*s.

Code context properties can be detect in two ways. The first method is by scanning an abstract syntax tree (AST) for specific code structures. The AST is created from one java file and has the limitation that it only contains information about the Java code defined in this file. In other situations we have to retrieve information from types that are not declared in the local file, e.g. when determining if a method is part of an override relation or not. In this case we use a method of creating an AST in combination with a symbol resolver. The symbol resolver is used to determine where external types are declared [13]. When the external type has been resolved an AST can be generated from the external types and be used again to retrieve the necessary information.

We have used *JavaParser* 0.6.0 library to create an AST from our Java code and perform analysis on this AST. The advantage of this library, over other parsers we have evaluated, is that it has good symbol resolving support present by the *JavaSymbolSolver* library that integrates seamlessly with the *JavaParser* library [52].

5.4.2 Data flow analysis

We have to consider preservation of name binding, data flow and control flow when performing an extract method refactoring [45, 49]. Preservation of name binding and control flow could be solved relative straight forward for the code context properties we identified for extract method refactoring (appendix C). In case of data flow preservation we needed a more complex algorithm that could determine those variables, including their names, that are accessed and/or modified. This detection should be based on a code fragment that is defined by start and end line in a specific method and class. We did not discover a public library that could provide us the required functionality. The lack of a library made us decide to implement a data flow analysis packages ourselves based on an algorithm which uses Boolean flags and expressions to determine the data flow of each variable individually in a method [28].

The implementation of this algorithm can be found in our code base in package *analysis.dataflow*.

5.4.3 Extendibility requirement

We introduced in 3.3.3 the non-functional requirement that future extension of our prototype with new refactorings should be easy. We achieved this by generalizing the detection of code context properties and the instantiation of refactoring guidance. Extending the prototype with new refactorings or expanding existing refactorings can be easily achieved by providing a new or updated RAG and additional *ContextDetector* classes.

Our prototype determines all necessary *ContextDetectors* based on information from the RAG. This is done by using names in the RAG for the CCPD-functions that have equal named counterpart classes that derive from *ContextDetector*. We use Java reflection [38] to determine if all required .class equivalents are available. If all needed *ContextDetectors* are present then we start traversing the RAG to determine the advices for the final refactoring guidance.

5.5 Identifying Code Context Properties

As explained in the previous sections we have to identify for each specific refactoring the relevant advices that could be potentially part of the generated refactoring guidance. Each of these advices has to be associated with a specific CCP. In this section the process followed to identified advices and CCPs is explained and we present a preliminary overview of identified CCPs for the refactoring *rename method*.

The same process was followed for identifying code context properties for the extract method refactoring. The outcomes of this analysis is added as an appendix in: 12 *Appendix B – Identified Extract Method* .

The process below describes how to determine relevant code context properties and their associated advices:

1. Fowler's Mechanics [19] are used as a starting point. The described mechanics in Fowler's book serve as the expert knowledge on how to perform manual refactorings of code.
   Mechanics contain stepwise instructions on how to manually refactor code. These mechanics describe also examples of alternative refactoring scenarios. This descriptions served as a basis for some simple code samples that were used to execute manually the described main and alternative mechanics.
2. The mechanics found in -1- are discussed with colleagues and experiments are performed on the created code samples to identify possible alternative mechanics that might be missing in the described mechanics.
3. Additional work is studied to discover alternative mechanics for our selected refactoring that were not yet identified in step 1 or 2. In our case we studied cases related to *rename method* and *extract method* [16, 45, 49].The already found mechanics are extended based on literature.
4. The steps in the mechanics identified in step 1-3 are analyzed to determine which code context properties are related to certain steps . For each identified CCP and its related steps an advice was formulated.
5. For each advice and its related context property found in 4 it was determined if an advices is an plain instruction, recommendation or warning.
6. Code context specific parts in the text of an advice were identified, e.g. where can we use namings from code to make the advice more concrete for a student.
7. We determined if a preferred order of advices would be needed and which code context properties should be grouped to form mutual exclusive code context properties.
8. Based on the input of 4-7 a RAG can be created.

We do not pretend that our findings of code context properties and advices are complete for *rename method* and *extract method*. The outcomes are merely a first. Future work could extend the presented cases. The RAGS that are the result of this process are presented in chapter 6 - *Evaluation*.

5.5.1 Rename Method

Situation: *The name of a method is not self-explaining its purpose*
Activity: *Change the name of the method so other engineers understand its purpose*

We introduce the identified code context property according the following structure:

- An overview of scenarios that we can address by a specific Advice. For each scenario we describe between brackets the name we will use to refer to a code context property that will be associated with this scenario.
- After this overview we describe:
    - How the CCP can be recognized in a code context.
    - Why the CCP is relevant.
    - A proposal for the content of the advice associated to the CCP.
    - The information that should become a hash-tagged value in the advice template.

An overview of used advice templates for the identified CCPs in our prototype is presented in chapter 13 - *Appendix C - CCPs Mapped to Advice Templates*.

5.5.1.1 Identifying refactoring scenarios

Here the identified scenarios are summarized. Between brackets we mention the name that is used to refer to a specific CCP.

Fowler identifies three scenarios when renaming a method [19]:

1. Single declaration of a method signature. [*Single Declaration*]
2. Multiple declaration of a method signature. [*Multiple Declaration – Override*].
    a. The method signature being renamed in a class has the same method signature implemented in one or more super classes of its inheritance hierarchy.
    b. The method signature that is being renamed in a class has one or more sub classes that have the same method signature implemented. This specific case has not been implemented in our prototype.

Additional scenario's we identified:

1. Method being renamed has a method signature that is defined public in an interface [*Multiple declaration – Interface*]
2. Method being renamed has overloaded methods [*Method Overload*].
3. For method being renamed overrides exist that are not marked with *@Override* annotation [*Override without annotation*]

5.5.1.2 CCP - Single Declaration

A single declaration CCP is detected when the method signature of the refactoring subject is only appearing once in all class definitions found in the inheritance hierarchy.

In this case the single method declaration can be renamed directly without special attention.

The content of the advice can be to rename the method, compile the project and solve any 'unresolved name' compiler error. We can see this advice as a general instruction for renaming a method. This advice can be seen as a 'default' advice as mentioned in section 5.2.3, because it will appear in any refactoring guidance at a certain point.

From the code context the original method name and class name might be extracted to be used in the advice template.

5.5.1.3 CCP - Multiple Declaration – Override

A 'multiple declaration – override' CCP is detected when the method signature of the refactoring subject is appearing multiple times in all class definitions found in the inheritance hierarchy.

In this specific case we want to create awareness that the method being renamed in a class is part of a inheritance tree that contains methods with the same signature. It should be made clear that only renaming the method in this class and not in the super and sub classes can have an effect on the functional behavior of the code.

The advice can be presented as a warning that in the inheritance tree method signatures are present that equal the method being renamed. It could be suggested to also rename those other methods, because otherwise the behavior of the code might change.

The template advice can define a keyword to hold a list of super and/or sub classes where the same method signature is present.

5.5.1.4 CCP - Multiple Declaration – Interface

A 'multiple declaration – Interface' CCP is detected determining if the method signature of the method being renamed is also present in an interface definition on which the encapsulating class has a dependency on directly or indirectly via another class in the inheritance tree

Renaming the method signature of a class where this method has been defined within an interface can be seen as a special case of the multiple declaration scenario as described by Fowler and from that perspective might not be treated as a separate case. We think that isolating this case is relevant because there is a risk involved when just renaming the method signature in the interface definition along together with the signature in the implementing class. Especially in those cases where the interface is exposed public. A change in the interface might have the effect of breaking dependencies to external packages that are not directly visible to the developer.

For this reason we think that at least there should be awareness of the possible side effect and provide an advice to the student that an alternative refactoring procedure can be relevant in this case.

The advice can be presented as a warning and instruct to create a new method with the new name and leave the original method in the interface definition with *@deprecated* added. To prevent code duplication the original methods should forward method calls to the newly added method signature.

For example, starting with the code below:

```
Public Interface IRESTSpecial
{
     Void renameMe();
}

class RestAdapter : implements IRestSpecial
{
   public void renameMe(){…}
}
```

Will result after the proposed refactoring procedure in:

```
Public Interface IRESTSpecial
{
     @deprecated
     void renameMe();
```

```
    void newName();
}

class RestAdapter : implements IRestSpecial
{
   public void renameMe(){
       newName();
   }

   public void newName(){…}
}
```

The advice template should hold parametrized information on the interface that is involved in this case.

5.5.1.5 CCP - Override without annotation

A 'Override without annotation' CCP is detected by determining the annotations that are defined with all overrides being detected in the class hierarchy. This could be done for both sub- and super classes.

From an understandability perspective and to prevent potential mistakes we think that every overridden method should have the *@Override* attribute added.

The advice can state that by using *@annotation* the compiler will help in detecting whether or not you are overriding a method and/or providing the right signature for that what you are trying to override. The advice can be presented as a recommendation while the attribute is not necessary for correct working of the code, but it can help in preventing potential mistakes.

The advice should hold parametrized input for which classes contain methods that are overridden with no *@Override* defined.

5.5.1.6 CCP - Method Overload

A 'Method Overload' CCP is detected there are methods with the same name as the method to be renamed in the same class

The understandability and maintenance argument for refactoring made us decide that method renaming should also be taken into account the occurrence of *method overload*. Method signatures with the same name in a class at least suggest that these methods are related to each other. Renaming one of these methods might in all probability also suggest renaming the other methods.

The advice can have the content as above and be presented as a recommendation to the user.

The advice template should contain parametrized keywords for holding a list of method signatures with the same name as the method being renamed and the location of the method signatures in the code.

6 Evaluation Results

The first result of our study is the theory presented in chapter 5. The theory describes a model and accompanying algorithm that can be used to generate refactoring guidance based on code context. In this chapter, we present: the preliminary evaluation results of the presented theory, the verification results of the developed software prototype, the student evaluation results of the presented concept and a discovered problem in a used theory on how to perform data flow analysis on a method.

First, we present the constructed RAGs for *rename method* and *extract method* in section 6.1. The CCPs, needed by the CCPD functions in the RAG, have been identified by following the process described in section 5.5 and are presented as part of our results. The concrete construction of the two presented RAGs evaluates if the presented theoretical model can be used for its intended ended purpose, storing the necessary information we need on concrete refactorings. We elaborate in section 6.1 also on some special constructions that we introduced in our RAG during construction. These constructions were needed to comply to the presented theory. The apparent limitations of our theory are addressed. The special constructions and apparent limitations are discussed later in 7 - Discussion & Future Work.

Second, we present the outcomes of the software prototype that was built as a proof-of-concept to generate refactoring guidance. Refactoring guidance has been generated for a number of predefined use cases. These results can be used to verify that the theoretical model and accompanying algorithm, presented earlier, generate expected results.

Third, we present the verifications that were performed on the software prototype for some functional and non-functional requirements of our software. The verification revealed an issue with the definition of liveness in the theory we use to perform data flow analysis. This issue is discussed in some detail, including the used solution.

We conclude the chapter with student evaluation results of the presented concept and generated refactoring guidance.

6.1 Evaluation of Theory

6.1.1 RAGs

The analysis of *rename method* and *extract method* refactoring procedures resulted in some specific CCPs that can be associates with specific Advices. For each of the identified CCPs, CCPD-Functions are defined that are used in the two RAGs we constructed for both *rename method* and *extract method*. In this section we present the two RAGs and necessary CCPDs. We discuss shortly some decisions that were made to make the RAGs comply with our definitions and where we have not been able to match completely with the presented theory. In chapter 7- *Discussion & Future Work*, we discuss in more detail if the presented solution is ideal and which possible improvements to the RAG could be investigated for future work.

As stated earlier the presented RAGs do not cover all possible cases for the mentioned refactoring. It is also up to future work to extend the presented cases.

6.1.1.1 Rename Method RAG

Table 1 list the CCPD functions we have identified for the rename method refactoring. There is a CCPD function for each CCP introduced in section 5.5. For each CCPD function a short description is given and

we name the context detector class that is responsible to implement the CCPD-function in our prototype.

Each identified CCPD-function (Table 1) is provided as parameter to the CCA-functions used to construct the RAG presented in Figure 14. The big ellipse vertices contain concrete advices. The template advices that are used to instantiate concrete advice can be looked up in *Appendix C - CCPs Mapped to Advice Templates*. The small ellipse vertices are explained later. They are a special type of advice that we introduced to bring our theory in practice.

Table 1 – Identified Code Context Property Detector function for rename method, a short description and corresponding name of context detector classes in the prototype's code base.

CCPD function	Description	Context Detector
SiD	Method declared only once	MethodSingleDeclaration
MD	Methods with same signature in class hierarchy	MethodMultipleDeclaration
ID	Method declared in public interface	MethodInterfaceDeclaration
SD	Method overrides a method of one of the super classes	MethodOverride
MO	Method overloaded in same class	MethodOverload
MA	Method can have @Override annotation	MethodOverrideNoAnnotation

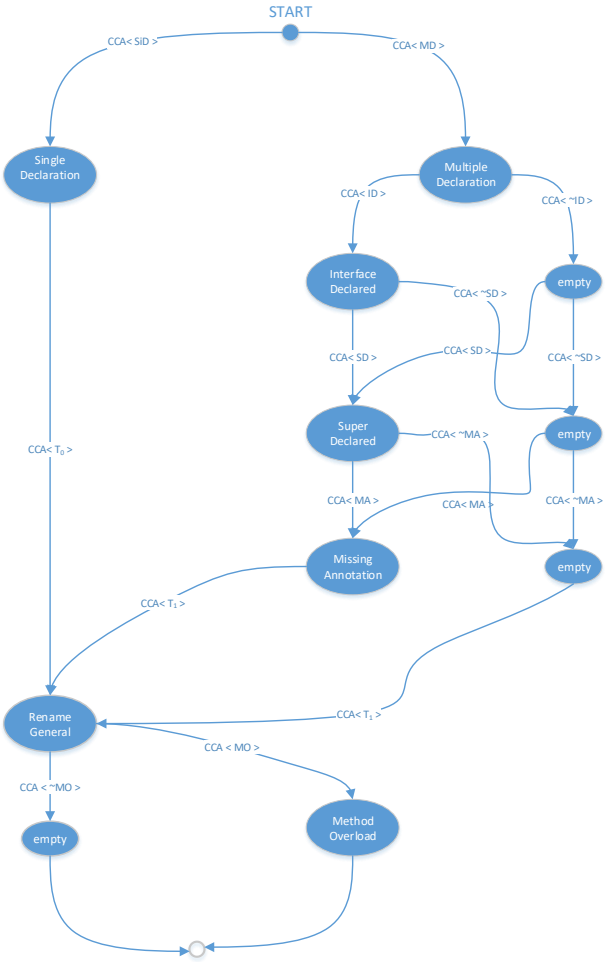


Figure 14 RAG - Rename Method

A first step towards composing the RAG was identifying which CCPs form so called code context property groups (CCPG), as presented in chapter 5.2. We can see in the RAG that the CCPs belonging to CCPD functions *SiD* and *MD* form a CCPG. All other CCPs form CCPGs that consist of two members : the CCP itself and its negated counterpart.

The negated CCPD-functions are introduced to comply with the restriction we placed on the RAG that always one of the outgoing edges from a vertex should contain a CCA-Function that evaluates to a next advice. This introduction of negated CCPD-functions led to the definition of an *empty advice* vertex. We consider each empty advice as unique so that the restriction holds for the RAG that the result of each CCA-Function is associated to one specific advice.

The need for empty advices is because we would otherwise violate the restriction that each CCA function that has the same CCPD function as a parameter must lead to the same advice. A simple solution could be by replacing the CCA-function, with the negated CCPD function as parameter, by the CCA-function that equals the other incoming edges of the next advice, but then then we would violate the mutual exclusive restriction again of the previous advice. It would also assume in this case that the next advice is always true if the previous advice was false. Consider advice 'multiple declaration' in the RAG applying the above would result in the two outgoing edges "CCA< ID>" and "CCA< SD >". These are clearly not mutual exclusive and also introduce a possible lock when evaluating the tree, what to do when the code context properties are present that indicate that the method has not been declared in the interface and the method has not been overridden from a parent class. The empty advices offer us a simple way of adding new decision points after each CCA that contains a negated CCPD function. Secondly, the originally restrictions we placed on our model still holds in this case.

Lastly, we decided if an ordering of advices, that can be instantiated from the given CCPs, would be preferred or not. We considered the "Method Overload"' advice as a hint that should be given after the rename of the actual method has taken place. We assume that an engineer would most probably proceed with renaming overloaded methods when done with renaming the initially selected method. We also assume in our RAG that the general advice on how to rename can be done after processing advices about annotation, super declarations and public interfaces. Although we do not have a strict reason for this ordering, variations might be considered.

In the next section we demonstrate that we were able to construct successfully a RAG for extract method in the same way as presented here.

6.1.1.2 Extract Method RAG

The meaning of elements in the depicted RAG in Figure 15 are the same as those used for *rename method* RAG. Additionally, we have added names for each code context property group in the right side line. An explanation of the identified CCPs in a bit more detail is given in *Appendix B – Identified Extract Method* .

We will also explain here some striking issues we encountered when constructing this RAG. We will discuss later in chapter 'discussion & future work' what possible solutions can be to overcome the problems we found.

Table 2 Identified Code Context Property Detector function for Extract Method, a short description and corresponding name of context detector classes in the prototype's code base.

CCPD function	Description	Context Detector
NH	Local variable hides class field	MethodExtractNameHiding
ZA	Zero arguments must be passed to extracted code	MethodExtractNoneArguments
SA	Single argument must be passed to extracted code	MethodExtractSingleArgument
MA	Multiple arguments must be passed to extracted code	MethodExtractMultipleArgument
ZR	Extracted code should return no result	MethodExtractNoneResults
SR	Extracted code should return single result	MethodExtractSingleResult
MR	Extracted code should return multiple results	MethodExtractMultipleResult
CRet	Conditional return present in extracted code	MethodExtractControlReturn

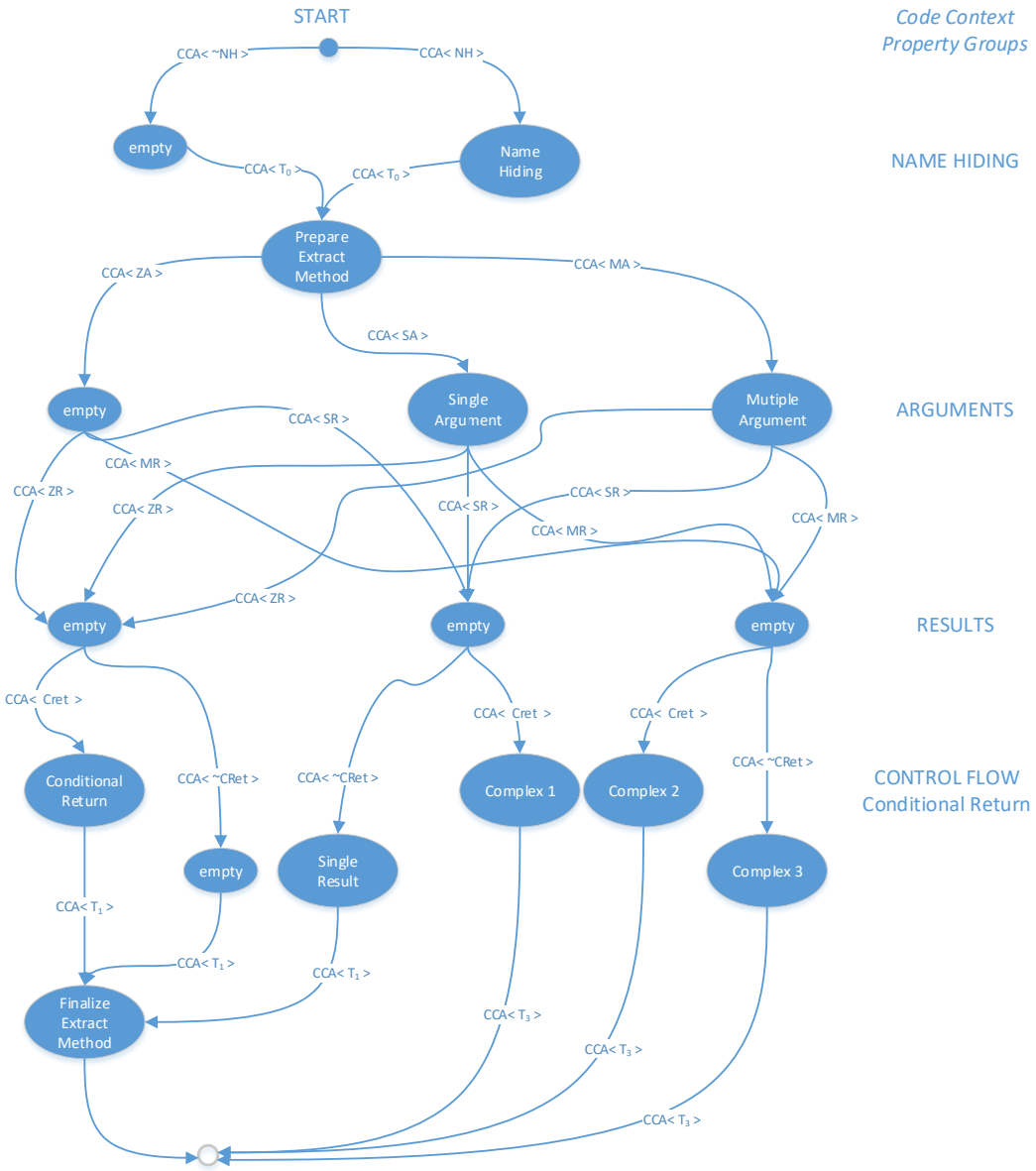


Figure 15 RAG - Extract Method

The extract method RAG (Figure 15) contains a few notable items that need explanation. On the right side of the figure we have added the code context property groups (CCPG) that have been identified for this refactoring. On the same height in the graph CCA-Functions are depicted that take CCPD-functions which are associated with CCPs that belong to the same CCPG. The RAG shows at the position of the "Control Flow" CCPG that the restriction stated in the theory 'each CCA function that has the same CCPD function as a parameter should lead to the same advice', no longer holds. We will discuss this apparent contradiction with our presented theory in section 7.1.

Another notable item in the RAG are the "Complex *n*" advices. We have mentioned these Advices "Complex *n*", because there is not a straight refactoring solution in these cases. For example, the path in the RAG leading from "Multiple Argument" to "Complex 3" shows that multiple arguments should be return by the extracted method. It is not possible to simply return the arguments. There is more investigation needed to determine these complex cases.

6.1.2 Verification of Theory by Proof-of-Concept

A software prototype is built to verify that that our theory, presented in chapter 5, can actually be used to automatically generate refactoring guidance. This was done by running multiple pre-defined scenarios, of which two are described here.

For these verification scenarios we defined some code fragments that contain beforehand known code context properties which are supported by our prototype. This pre-defined code fragments can be selected directly in the prototype.

For each of the code fragments we reasoned what the expected outcome for a specific refactoring procedure should be in the form of a textual refactoring guidance report. Refactoring guidance was generated by using our prototype for these code fragments. The generated results were compared with the expected outcome to judge if the generation of refactoring guidance behaves as expected.

Two example scenarios that were ran are described here. They are based on code sample API_Rename.java and ExtractMethod.java:

1. Renaming of method *getAccountName* in class *API_SpecialImplementation* of pre-defined source code *API_Rename.java*. This case exposes the method to be renamed via a public interface declaration. For the method to be renamed also has overrides exists in the base class *API_Implementation* and overloaded method signatures are present in the code. Finally, the code did not use the @Override consistently in the code.
2. Extracting code on line 45 from method *longMethod()* in class *EM,* file *ExtractMethod.java*. The variable *firstName* hides a field with the same name. The extracted method depends on two input variables and changes one variable that is used later on in the source method as a parameter of a system call.

Appendix E – Example of a Generated Refactoring shows the outcomes of the generated refactoring guidance.

6.2 Software Verification

6.2.1 Functional

To verify if some specific functionalities in our prototype behave as expected, 92 unit tests have been implemented. These unit tests verify specifically:

- Correct detection of specific code context properties for each implemented context detector. This assures that each CCA-function and its CCPD-function in the presented RAGs can be successfully implemented.
- Correct generation of refactoring guidance based on: java code fragment, RAG and exactly one predefined code context.
  This assures that the basic generation of the refactoring guidance is properly working.
- Correct working of data-flow analysis for a selected set of fundamental forms of expressions (Watt, 2006), which are: Literals, Constructions, Function calls, None nested Conditional expressions, Variable access
  This assures that data-flow analysis works for simple intra method purposes.

6.2.2 Non-Functional – Extendibility Use-Cases

We defined a non-functional requirement in chapter 3 to design a prototype that can easily extend or introduce new code refactorings that might be added for future research. Our prototype has been designed to have a generic refactoring guidance generation algorithm, which allows us to add or extend future code refactorings with minimal change to existing classes .

We verified the extendibility requirement by demonstrating here what was needed to successfully introduce *extract method* refactoring in our prototype. We only had to follow the six steps below to extend our prototype's functionality. The italic classes refer to class names in the prototype code.

1. A new RAG was described for *extracted method* in *AIG_ExtractMethodGeneration.java*
2. Each CCA-function of the RAG has been implemented by its own class *MethodExtract<XXX>.java* which is derived from *ContextDetector*.
3. Each class derived from *ContextDetector* uniquely identifies itself by an enum of type *CodeContextEnum*. This unique enum value corresponds with values in the RAG definition of point 1. *CodeContextEnum.java* is extended with the newly defined enums.
4. The dataflow analysis algorithm has been added to the project as package *analysis*. This step is only necessary when additional analyzers are needed by the *ContextDetector* classes.
5. Extend class *context.ContextConfiguration* with a method that provides access to the new dataflow analyzer. *context.ContextConfiguration* serves as a container *ContextDetectors* can access these analyzers in a generic way.
6. *ContextDetectorSetBuilder* was extended with the method *BuildExtractMethodContextDetectors()*. This method is responsible to instantiate objects and analyzers in a correct order for this specific refactoring.

6.2.3 Definition of Liveness

The algorithm for intra method data-flow analysis [28] revealed an inexactness in the workshop paper related to the definition of when a variable is live. This algorithm divides the method in three regions: the code to be extracted (extracted region) and a region before and a region after (after region) this extracted code. Liveness of a variable in the region is defined such that a read access occurs before a certain write access. To determine if the result of a variable should be returned from the extracted region, the check is done if a variable is life in the after region and that the same variable has been written in the extracted region. The definition of liveness given by Juillerat suggests that a read of a specific variable in the after region should always be followed by a modification of this same variable. This would mean that when a variable is modified in the extracted region and only read in the after

region the variable would not be marked live, so should not be returned by the extracted region to the after region. This outcome is clearly false. To overcome this issue we adapted our algorithm to use the definition of liveness as being given by Nillsson-Nyman: *"A variable is live if its assigned value will be used by successors in the control-flow graph. If the variable is assigned a new value before the old value has been used the old assignment can be considered unnecessary"* [45].

6.3 Student Evaluation

After iteration two and three we evaluated the concept and generated refactoring guidance of our prototype. Iteration two was concluded by individual interviews. Iteration three was concluded by group discussions. The outcomes of these interviews have been summarized and grouped into three categories: how do students refactor, positive perceived aspects of the tool and future improvements. We are aware that the number of interviewed students is too small to draw generalized conclusions, but we think the summarized answers still are valuable enough to present here for providing possible directions in future research.

How do students Refactor

All students we interviewed were able to give a definition of refactoring that made clear it was about improving the readability and maintainability of the code or software design. How refactoring was actually performed and judging what the effects of their actions might be differed a lot. Every student explained they would determine based on the content of a method what might be a good new name for the method. After this, they would either proceed with a manual refactoring or automatic refactoring. How a student would proceed seemed to be depending on the experience students gained at their internships after the second year of their study. Those students that had their internship at companies that followed a strict process and where senior engineers reviewed their code seemed to follow a much stricter refactoring process according to their description.

All students confirmed that if the question about their refactoring strategy had been asked in their second year of their study, they would probably have answered that they followed a trial-and-error strategy: rename the method manually, compile and hope that it works.

For those who used automatic refactoring tools, they were aware that this might not always lead to the desired result and gave examples of experiencing: compiler errors, failing test cases or unreadable code. The students also recognized those cases where they found out later that their software projects would not work functionally anymore as expected after refactoring changes had been made either manually or automatically.

To the question how students use the automatic refactoring tools the general answer was, press OK and continue followed by either some manual tests or unit tests if they were available. Additional information or preview possibilities of automatic tooling were in most cases not used. Also students could not explain what options offered by the refactoring tools actually did or meant. For most students we interviewed it was hard to tell what could possibly go wrong in the rename method, besides that the method would not compile.

Positive Perceived Aspects

New insights were gained by most students when using the generated refactoring guidance like:

- Better understanding of the structure and existing dependencies in the code. "…*It gives insight in your code on a much more syntactic and conceptual level than when you were performing a refactoring manually without guidance or with an automatic tool…" (anonymous)*
- Understanding of the public interface dependencies by external projects and how @deprecated can play a role in managing changes on public interfaces.
- Better understanding of specific language features like: @deprecated, @override
- The direct instructions on how and why to perform specific steps in a refactoring made them think more about what is actually going during the refactoring procedure. Refactoring is becoming a more conscious activity with the tool *"…With automatic tooling you just press ok and experience possible problems of your refactoring much later. Now it lets you think about it…" (anonymous)*
- Positive that you get concrete hints to potential risks and how they could be resolved.
- The combination of why and how in the instructions.

In the last group evaluation students rated the outcomes of our prototype as 'very useful'. The general consensus was in this groups that especially in the first years of their education, when refactoring is completely new, it seemed to them as a tool that could be most useful. This seems to match with a pointer we got in the first individual interviews where students indicate that especially in the second year they were working in trial-and-error mode and hot not a clear guidance on what they were actually doing.

Future Improvements

In the last group evaluations the following improvements have been suggested by the students:

- The current ordering of refactoring steps, especially in rename method, are not always intuitive.
- The risk identifications are explained in the output on why it is a risk. It would also be very interesting to know what the refactoring steps bring as a result. How are they improving the code?
- Spelling and structure of sentences of the instantiated refactoring procedures.
- Some of the instructions are too much detail on how to perform the code transformations, it might have the effect of students not thinking anymore. An improvement might be to hint to a desired end result than exactly stating what to do. Example: state 'The extracted method should return variable b as bool' than 'Add instruction *return b;* at the end of the extracted method'.
- Make the output of the tool configurable based on seniority level of student. This might introduce or hides certain steps in the instantiated refactoring procedures.
- Make sure the architecture of the tool is open for extension. It would be interesting if students could add their own context detectors with rules for detecting code smells of which they think they are important to solve.

7 Discussion & Future Work

7.1 Discussion

7.1.1 Results

The results presented in chapter 6 demonstrate that our proposed theory, to generate refactoring guidance constructed out of advices that are uniquely related to CCPs, is feasible. We are able to generate context based refactoring guidance given: a RAG for a selected refactoring and a refactoring subject together with its code context.

Expert knowledge was studied to determine how to manually perform *rename method* and *extract method*. The acquired knowledge is used to construct a RAG for each specific refactoring. These RAGs were evaluated by having our prototype successfully generate refactoring guidance in different code contexts. However, some special constructions were needed to construct the RAGs and apparent limitations appeared. In Section 7.1.2 - *Limitations* we elaborate on this observed issues and suggest possible solution directions.

The gathered expert knowledge largely depends on the work of Fowler [19] and a study on *extract method* [49]. For our study these sources prove sufficient, because we did not intend to make complete RAGs for both refactorings. Nevertheless, we took some time to extend the cases from literature with input from our peers. It is notable that expert knowledge could easily be extended with new scenarios for both refactorings. This suggests at least that described scenarios in current literature are not complete and could be deepened further. The same suggestion goes for the complex advices we have seen in the *extract method* RAG. There is no specific expert knowledge described in literature on best practices by professionals to solve these complex scenarios.

Our theory to automatically generate refactoring guidance is based on the detection of code context properties that are related to unique advices. This approach makes it possible to generate guidance for any piece of compilable Java code in contrast with model based solutions, like AutoStyle [39]. On the other hand, we cannot monitor if students are progressing towards a correct final solution. This limitation seems not to have influence on how students rate our concept and generated guidance. All interviewed students rated the guidance as ‘useful’ and indicated that the guidance provided them with new insights. This matches with the evaluation outcomes of a similar tool like FrenchPress [8]. This might be an indication that educational tools can be effective even without steering to a model solution

Another observation we did in the student evaluations is that the learning effect might go beyond our initial goal: increasing understanding of the refactoring processes. The effect of our generated guidance also helped some students to gain new insights in specific Java language constructs and understanding of the underlying software design. The better understanding of software is not really unexpected, considering that refactoring is about improving the design of code.

We also see an interesting parallel between professional software engineers and undergraduate SE students. Namely, the lack of: understanding the refactoring process, the correctness of the refactoring outcomes and understanding of the possibilities of the tools. These are all points mentioned in studies that looked into how software engineer use and experience automatic refactoring tools [18, 44, 60].

7.1.2 Limitations

The evaluation of our theory and prototype also revealed some limitations that we like to discuss here.

The evaluation of our theoretical model revealed a few issues that had to be solved by some special constructions: the empty advices and negated CCPD functions (Figure 14). These constructions seem to make the RAGs unnecessary complex. A simplification of the model seems to be possible by introducing composed CCA functions by Boolean operators: NOT (!) and AND (^).

As an example we can take the "CCA<~ND>" transition after the "multiple declaration" advice in the rename method RAG (Figure 14). This transition could be rewritten to "! CCA<ND>". The empty advice that is following behaves like an AND operator. So the path that now leads from 'multiple declaration' to 'Super Declared' could be redefined with Boolean operators as: "! CCA<ND> ^ CCA<SD>". The restriction in our theoretical model tells that a CCA function that takes the same CCPD function has one unique advice as a result. This could be resolved by redefined the restriction to: a Unique advice is related to logical equivalent CCA-Functions

This proposed change can also solve the issue we have seen in the *extract method* RAG. Here we have CCA functions that hold the same CCPD function, but result in multiple different advices. An example of this is given by the extract method RAG (Figure 15). where "CCA<CRet>" can lead to three Advices: "Conditional Return", "Complex 1" and "Complex 2". If we rewrite the transitions of the path leading to "Conditional Return" and "Complex 1", we see that this rewritten CCA evaluation leads to a valid result while the composed functions are not logical equivalent.

CCA<ZA> ^ CCA<ZR> ^ CCA<CRet>	→	"Condition Return"
CCA<SR> ^ !<CRet>	→	"Complex 1"

It would be interesting to further investigate this approach.

The last limitation we like to mention here is that at this moment we only evaluated the theory by verifying if the generated refactoring guidance, matches our expected outcome. This verification has been done with a limited number of scenarios and has not been used in real-life case scenarios.

7.1.3 Related Work

Chapter *Theoretical Background* mentions two educational tools that generate hints and feedback on how to refactor code to improve code style: FrenchPress [8] and AutoStyle [39]. How does our approach compare to these two tools?

The major difference is the kind of refactoring that we address. FrenchPress and AutoStyle generate hints and feedback aimed at code improvements which do not go beyond the method scope, for example rewriting a logical condition or an if-statement. The guidance we generate is based on the refactorings described by Fowler. Those refactorings lead in many cases to code changes that go outside the scope of a method and needs analysis of the code that is outside method or even class definitions.

Another difference is that we can control the order of advices in our refactoring guidance. These advices from together a group of instruction, risk identifications and code improvements which are all related to one and the same refactoring. FrenchPress and AutoStyle generate hints and feedback that are unordered and do not necessarily have any relation with each other, which might make them more incoherent as feedback to students.

Tools like FrenchPress and our prototype seem to offer more flexibility, than model based solutions like AutoStyle, to teach students in their own specific context. Any piece of code a student is working on

could be input to generate guidance. While we can work with the same set of rules for many pieces of code, this might suggest that it is less time consuming method than defining a model for each specific problem. In one study it is stated that most refactor tools cannot guarantee refactoring correctness, since formally guarantee is cost-prohibitive [53]. This might also suggest that modelling all possible solution paths is even not feasible. On the other hand, tools that do not monitor for progress to a correct final solution might only be effective for students that already possess sufficient basic skills in coding and refactoring. The evaluation outcomes of FrenchPress and our prototype at least suggest that students not necessarily need feedback that leads to one correct solution to appreciate a tool and learn from it.

7.1.4 Generalization

The presented theory to generate refactoring guidance seems to be generic enough to be used with other programming languages and maybe even other programming language paradigms .

Within the same paradigm it will be the content of advices that vary per language. The specific code context properties to be detected which are related to the advices can probably stay the same in many of the cases for several OO languages.

When looking at other language paradigms changes will be bigger. In this case other refactoring patterns apply than the ones we have been studying, therefore this will lead to other code context properties to detect and new advices to formulate.

In both cases the adaptations we have to make only apply to the content of the RAG model. The algorithm to generate refactoring guidance is language agnostic while it uses only the information stored in the RAG. So, this means that with the right code context property detection functions and advices we would be able to generate refactoring guidance for any programming language based on our proposed theory.

7.2 Future work

The results from this study, the discussion and the presented vision of a workflow of a possible future refactor guidance tool have revealed many interesting questions that might be input for future research related to our study. In this section, some suggestions for future work are presented.

We have seen that the expert knowledge available on refactoring strategies can be deepened further and has not been described or investigated for more complex. Interesting research would be to observe how experts refactor specific (complex) cases. These observations could be used to extend the presented refactoring cases.

As has been discussed the theory only has been verified by comparing the generated outcome with expected results for a limited number of cases. We are interested to see how our presented proof-of-concept actually works in real-life. Is the refactoring guidance still useful then? Does it indeed increase a better understanding of refactoring? Does the learnings go beyond increased understanding of refactoring? A case study in an educational context might give answers to these questions.

Subsequent to the real-life cases, a proper validation of the models could be done by comparing the generated refactoring guidance with for example guidance that a real life teacher might give to a student.

Before being able to evaluate the prototype in an educational context, we might need to improve the tool from a usability aspect. Some ideas are listed below:

- By adding specific questions to the advices we could refine and filter those advices that are relevant to the students. For example “are the parameters provided in the new method related to each other?” When the question is answered by yes, this could be a trigger to create advices for refactoring “Group object for parameters”.
- Make it possible to indicate for a student to indicate which advices to ignored. This would reduce the text that is shown now.
- Let students mark Advices that they have followed, to keep the overview
- Anticipation to answers of students. For example when a rename method is performed we could ask for the new name to be entered. An analyzer could be added to determine if the new name conflicts with the existing code base.
- A student model could be used to modify the content of the created advices.

The suggestion on using Boolean operators in our RAG could be a topic to invest. It would be interesting to see what the implications are of these changes and if it does lead to a simplified model. In this context we could also investigate if other models could be used that are more simple and effective.

From the generalization perspective we presented, it would be interesting to look if the claims we make are correct. Can we indeed use the presented model & algorithm in other programming language?

When specifically looking into detecting of code context properties for a language belonging to the same paradigm it might be interesting to investigate if CCPD functions can be specified by using a domain specific language (DSL). This would enable us to define the detection of specific CCPs in a generalized form and write an interpreter per language that can scan for specific code context properties based on the general definition.

Other interesting future research is related to the workflow we have presented in our vision. Is the workflow complete like this? Can other existing work be related to the presented steps and is the suggested work indeed useable in the context we have presented?

The last suggestion is to investigate if monitoring a students’ progress to a correct end solution can be included, without being dependent on predefined models or limiting the exploring possibilities of the generated guidance. An idea would be to extend each advice with clear step-by-step instructions that could be translated to code transformations. These transformations could be used to generate a dynamic model real-time that could be used to track the students’ progress in its specific context. The work of Schäfer et al. [49] could be inspiration on how to describe the proposed transformations.

8 Conclusion

This thesis describes the investigation on:

> *How to automatically generate refactoring guidance for Java code that is based on code context?*

The formulation of this main research question is based on the presented vision of an ideal refactoring guidance tool for undergraduate SE students. We suggest in this vision functional steps of a workflow that such a tool should support. One of the steps in this workflow is to provide student guidance on how to manually perform a refactoring. An analysis of existing studies showed that there is to the best of our knowledge no studies available that have looked into the generation of guidance out of arbitrary Java code on how to manually perform refactorings and augment this with warnings and hints to improve understandability of the code. In this study the focus was on the refactorings defined by Fowler.

The main research question has been divided into two sub-questions, for which concluding answers follow below.

> *How can we generate instructions on how to manually refactor Java code, that are adapted to code context?*

An answer is given by the theory presented on how to generate refactoring guidance that is based on code context, and including a proof-of-concept in the form of a software prototype. Within our theory, we generate refactoring guidance constructed out of advices. These advices can be: general instructions, recommendations or warnings. Each advice is associated with one specific code context property that can be present in the code context of a refactoring subject. Advices and associated code context properties are stored in a graph we call RAG. We describe an algorithm that instantiates advices in the presence of specific code context properties. The created advices are the building blocks of refactoring guidance that is code context based.

> *How should we, according to expert knowledge, perform refactorings rename method and extract method manually?*

The partial answer is given by the two concrete RAGs constructed for *rename method* and *extract method*. The advices stored in these RAGs are the result of analyzing available literature specifically on these refactorings. Advices were extended based on peer evaluation of the literature results. The process showed us that additional cases are easy to identify, so literature seems to cover mainly relative easy cases and is not to be considered complete. This is also emphasized by the fact that how to handle more complex refactor scenarios is missing in literature.

The software prototype has been used to generate refactoring guidance which is based on the constructed RAGs. The generated content was evaluated with undergraduate SE students, who are a target group to use this guidance. The evaluation results show us that the generated refactoring guidance, which is based on associating code context properties with Advices, is received positively by the students we have interviewed.

The concluding answer to our main research question is we presented a theoretical and practical solution on how to generate refactoring guidance based on code context, which is well received by our target group. However, to increase the number of supported refactorings and their level of complexity, more future research is needed.

9 References

[1] M. Abebe and C. Yoo. 2014. Trends, opportunities and challenges. *International Journal of Software Engineering and Its Applications* 8, 6, 299–318. DOI: https://doi.org/10.14257/ijseia.2014.8.6.24.

[2] N. J. Ahuja and R. Sille. 2013. A Critical Review of Development of Intelligent Tutoring Systems. Retrospect, Present and Prospect. *International Journal of Computer Science Issues* 10, 4, 39–48.

[3] J. S. Alghamdi, R.A.i Rufa, and S. M. Khan. 2005. OOMeter: A Software Quality Assurance Tool. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, 190–191. DOI: https://doi.org/10.1109/CSMR.2005.44.

[4] M. Ardis and D. Budgen. 2015. *Software Engineering 2014*. *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*.

[5] G. Bavota, A. de Lucia, M. Di Penta, R. Oliveto, and F. Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *The Journal of Systems and Software* 107, 1–14. DOI: https://doi.org/10.1016/j.jss.2015.05.024.

[6] P. d. Beer and S. Angelov. 2015. Fontys ICT, Partners in Education Program. Intensifying Collaborations Between Higher Education and Software Industry. In *Proceedings of the 2015 European Conference on Software Architecture Workshops - ECSAW '15*. ACM Press, New York, New York, USA, 1–4. DOI: https://doi.org/10.1145/2797433.2797468.

[7] J. Bennedsen and M. E. Caspersen. 2005. Revealing the Programming Process 37, 186. DOI: https://doi.org/10.1145/1047124.1047413.

[8] H. Blau and J.E.B. Moss. 2015. FrenchPress Gives Students Automated Feedback on Java Program Flaws. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '15*. ACM Press, New York, New York, USA, 15–20. DOI: https://doi.org/10.1145/2729094.2742622.

[9] D. M. Breuker, J. Derriks, and J. Brunekreef. 2011. Measuring static quality of student code. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education - ITiCSE '11*. ACM Press, New York, New York, USA, 13. DOI: https://doi.org/10.1145/1999747.1999754.

[10] Cambridge University. 2018. *Cambridge Dictionary* (2018). Retrieved October 14, 2018 from https://dictionary.cambridge.org/.

[11] O. Chaparro, G. Bavota, A. Marcus, and M. Di Penta. On the Impact of Refactoring Operations on Code Quality Metrics. DOI: https://doi.org/10.1109/ICSME.2014.73.

[12] J. Chen and S. Huang. 2009. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software* 82, 6, 981–992. DOI: https://doi.org/10.1016/j.jss.2008.12.036.

[13] K. Cooper and L. Torczon. 2011. *Engineering a Compiler* (2nd). Morgan Kaufmann.

[14] V. Dagienė, C. Schulte, and T. Jevsikova, Eds. 2015. *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '15*. ACM Press, New York, New York, USA. DOI: https://doi.org/10.1145/2729094.

[15] T. Ekman, M. Schäfer, and M. Verbaere. 2008. Refactoring is not (yet) about transformation. In *Proceedings of the 2nd Workshop on Refactoring Tools - WRT '08*. ACM Press, New York, New York, USA, 1–4. DOI: https://doi.org/10.1145/1636642.1636647.

[16] R. Ettinger and M. Verbaere. Untangling. A Slice Extraction Refactoring, 93–101. DOI: https://doi.org/10.1145/976270.976283.

[17] F. Fontana, P. Braione, and M. Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *JOT* 11, 2, 5:1. DOI: https://doi.org/10.5381/jot.2012.11.2.a5.
[18] F.A Fontana and M. Zanoni. 2017. Code Smell Severity Classification using Machine Learning Techniques. *Knowledge Based Systems* 128, 43–58. DOI: https://doi.org/10.1016/j.knosys.2017.04.014.
[19] M. Fowler. 1999. *Refactoring: Improving the Design of existing code*. Addison-Wesley Professional.
[20] R. Haas and B. Hummel. 2015. Deriving Extract Method Refactoring Suggestions for Long Methods. In *Lecture Notes in Business Information Processing*. Springer International Publishing, 144–155. DOI: https://doi.org/10.1007/978-3-319-27033-3_10.
[21] B. Heeren and J. Jeuring. 2014. Feedback services for stepwise exercises. *Science of Computer Programming* 88, 110–129. DOI: https://doi.org/10.1016/j.scico.2014.02.021.
[22] Hewlett Packard Enterprise. 2017. *Agile is the new normal. Adopting Agile Project Management* (2017). Retrieved February 9, 2019 from https://softwaretestinggenius.com/docs/4aa5-7619.pdf.
[23] IBM. 2017. *Eclipse*.
[24] G. Jay, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft, and C. Ward. 2009. Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship. *JSEA* 02, 03, 137–143. DOI: https://doi.org/10.4236/jsea.2009.23020.
[25] JetBrains. 2017. *IntelliJ*.
[26] C. Jones and O. Bonsignour. 2011. *The Economics of Software Quality*. Addison-Wesley Professional.
[27] D. Jönsson. 2013. detecting code smells in educational context.
[28] N. Juillerat and B. Hirsbrunner. 2007. *Improving Method Extraction: A Novel Approach to Data Flow Analysis Using Boolean Flags and Expressions*. *Workshop on Refactoring Tools*, Berlin, 48–49. Retrieved from.
[29] S. H. Kannangara and W.M.J.I. Wijayanake. 2015. An Empirical Evaluation of Impact of Refactoring on Internal and External Measures of Code Quality. *IJSEA* 6, 1, 51–67. DOI: https://doi.org/10.5121/ijsea.2015.6105.
[30] A. Kaur and M. Kaur. 2016. Analysis of Code Refactoring Impact on Software Quality. *MATEC Web of Conferences* 57, 2, 2012. DOI: https://doi.org/10.1051/matecconf/20165702012.
[31] H. Keuning. 2014. *Strategy-based feedback for imperative programming exercises*. Master. Open Universiteit Nederland.
[32] H. Keuning, B. Heeren, and J. Jeuring. 2017. Code Quality Issues in Student Programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '17*. ACM Press, New York, New York, USA, 110–115. DOI: https://doi.org/10.1145/3059009.3059061.
[33] H. Keuning, J. Jeuring, and B. Heeren. 2016. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. DOI: https://doi.org/10.1145/2899415.2899422.
[34] M. Kim, T. Zimmermann, and N. Nagappan. 2014. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IIEEE Trans. Software Eng.* 40, 7, 633–649. DOI: https://doi.org/10.1109/TSE.2014.2318734.
[35] Y.B.D. Kolykant. 2005. *Students' Alternative Standards for Correctness*. ACM.
[36] R.J LeBlanc and T. B. Hillburn. 2005. *Computing Curricula 2005*.
[37] M. Mäntylä. 2003. *Bad smells in software*. *A taxonomy and an Empirical Study*. Master. University of Technology. Department of Computer Science and Engineering, Helsinki.

[38] G. McCluskey. 1998. *Using Java Reflection* (1998). Retrieved April 12, 2018 from https://www.oracle.com/technetwork/articles/java/javareflection-1536171.html.

[39] J. Moghadam, R. Choudhury, H. Yin, and A. Fox. 2015. AutoStyle. Toward Coding Style Feedback at Scale. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale - L@S '15*. ACM Press, New York, New York, USA, 261–266. DOI: https://doi.org/10.1145/2724660.2728672.

[40] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba. 2014. Making refactoring safer through impact analysis. *Science of Computer Programming* 93, 39–64. DOI: https://doi.org/10.1016/j.scico.2013.11.001.

[41] E. Murphy-Hill. 2000. *Programmer Friendly Refactoring Tools*.

[42] E. Murphy-Hill. 2010. *An Interactive Ambient Visualization for Code Smells*. ACM, New York, NY.

[43] E. Murphy-Hill and G. C. Murphy. Peer interaction effectively, yet infrequently, enables programmers to discover new tools, 405. DOI: https://doi.org/10.1145/1958824.1958888.

[44] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it, 287–297. DOI: https://doi.org/10.1109/ICSE.2009.5070529.

[45] E. Nilsson-Nyman, G. Hedin, E. Magnusson, and T. Ekman. 2009. Declarative Intraprocedural Flow Analysis of Java Source Code. *Electronic Notes in Theoretical Computer Science* 238, 5, 155–171. DOI: https://doi.org/10.1016/j.entcs.2009.09.046.

[46] Oracle. 2017. *The Java Tutorials*. *Language Basics* (2017). Retrieved October 21, 2018 from https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html.

[47] H. Passier. 2016. Maintaining Unit Tests During Refactoring. DOI: https://doi.org/10.1145/2972206.2972223.

[48] N. Sae-Lim. 2017. How Do Developers Select and Prioritize Code Smells? a Preliminary Study. DOI: https://doi.org/10.1109/ICSME.2017.66.

[49] M. Schäfer, M. Verbaere, T. Ekman, and O. d. Moor. 2009. Stepping Stones over the Refactoring Rubicon. In *ECOOP 2009 – Object-Oriented Programming*, S. Drossopoulou, Ed. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 369–393. DOI: https://doi.org/10.1007/978-3-642-03013-0_17.

[50] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Fifth European Conference on Software Maintenance and Reengineering*, 30–38. DOI: https://doi.org/10.1109/.2001.914965.

[51] S. Singh and S. Kaur. 2017. A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal. DOI: https://doi.org/*10.1016/j.asej.2017.03.002.

[52] N. Smith, D. van Bruggen, and F. Tomassetti. 2017. *JavaParser: Visited*. *Analyse, transform and generate your java code base*.

[53] G. Soares. 2010. Making program refactoring safer. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. ACM Press, New York, New York, USA, 521. DOI: https://doi.org/10.1145/1810295.1810461.

[54] S. Stuurman, H. Passier, and E. Barendsen. 2016. Analyzing students' software redesign strategies. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research - Koli Calling '16*. ACM Press, New York, New York, USA, 110–119. DOI: https://doi.org/10.1145/2999541.2999559.

[55] E. R. Sykes. 2003. An intelligent tutoring system prototype for learning to program Java. In *Proceedings 3rd {IEEE} International Conference on Advanced Technologies. DOI: https://doi.org/*10.1109/ICALT.2003.1215208.

[56] G. Szoke. 2015. *FaultBuster: An Automatic Code Smell Refactoring Toolset*. IEEE, Piscataway, NJ.

[57] R.A Tayde, G.B Regulwar, and Nimbokar K.G. 2012. Impact of Refactoring on Software Quality Factors. *InternatIonal Journal of Computer Science and technology* 3, 4.

[58] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. 2008. JDeodorant: Identification and Removal of Type-Checking Bad Smells, 329–331. DOI: https://doi.org/10.1109/CSMR.2008.4493342.

[59] D. Turk, F. Robert, and B. Rumpe. 2005. Assumptions Underlying Agile Software-Development Processes 16, 4, 62–87. DOI: https://doi.org/10.4018/jdm.2005100104.

[60] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. 2012. Use, disuse, and misuse of automated refactorings. DOI: https://doi.org/10.1109/ICSE.2012.6227190.

[61] S. A. Vidal, C. Marcos, and J.A Díaz-Pace. 2016. An approach to prioritize code smells for refactoring. *Autom Softw Eng* 23, 3, 501–532. DOI: https://doi.org/10.1007/s10515-014-0175-x.

[62] A. Vihavainen, M. Paksula, and M. Luukkainen. 2011. Extreme apprenticeship method in teaching programming for beginners, 93. DOI: https://doi.org/10.1145/1953163.1953196.

[63] E. S. Wiese, M. Yen, A. Chen, Lucas A. Santos, and A. Fox. 2017. Teaching Students to Recognize and Implement Good Coding Style. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale - L@S '17*. ACM Press, New York, New York, USA, 41–50. DOI: https://doi.org/10.1145/3051457.3051469.

[64] D. Wilking, U. F. Kahn, and S. Kowalewski. 2007. An Empirical Evaluation of Refactoring. *e-Informatica Software Engineering Journal* 1, 1, 27–42.

10 Personal Reflection

Terugblikkend op het afgelopen afstudeertraject, kan ik stellen dat het me veel nieuwe inzichten heeft gebracht in wat wetenschappelijk onderzoek is en wat er bij komt kijken om dit zelfstandig uit te voeren. Wat daarbij het meest me zal bijblijven is dat de vorming van het uiteindelijk idee tijd nodig heeft om te rijpen en de tijd die nodig is voor het overbrengen van ideeën in begrijpelijke teksten voor nieuwe lezers. Vergeleken met andere trajecten in de studie was dit een lange periode met veel nieuwe uitdagingen, die het soms moeilijk maakte om dit volledig onder de controle te houden wat betreft tijd. Ik geef nu aan mijn studenten mee, die een wetenschappelijke studie nastreven, om dit zo snel mogelijk op te pakken. Het combineren van een studie met de onzekerheden van privé en werkleven maken het niet altijd even makkelijk om je aan een vastgestelde planning te houden.

Het grote verschil van het afstudeerproject met eerdere onderdelen in de studie is dat het onderwerp waaraan gewerkt wordt niet langer meer duidelijk is afgebakend. Het is nieuw om voor jezelf duidelijk te krijgen het einddoel helder te stellen. Om dit mogelijk te maken zijn grotere blokken tijd nodig om de focus te vinden, waarbij dieper nagedacht kan worden over de vraagstukken die er liggen en helder de verkregen ideeën op papier te zetten. Communiceren in de Engelse taal, waar ik redelijk wat ervaring mee heb in een informele setting, blijkt daarbij dan ook ineens een remmende factor in het schrijven en duidelijk communiceren.

Als ik terugkijk naar hoe de keuze van mijn onderzoeksonderwerp tot stand is gekomen, heb ik me soms afgevraagd of ik toch beter niet een onderwerp had kunnen kiezen wat direct aan had gesloten op een lopend onderzoek. Het had het afbakenen van het onderzoeksonderwerp misschien makkelijk kunnen maken. Toch denk ik dat mijn keuze uiteindelijk juist is geweest. Het onderwerp is een idee, dat ik ooit al eens geopperd tijdens mijn eerste mondeling in het pre-master traject voor het vak Software Engineering. Refactoring is een onderwerp waar mijn interesse ligt en ik heb de mogelijkheid gehad om aan het begin heel breed naar dit onderwerp te kunnen kijken. Vanuit mijn onderwijsachtergrond heb ik daar een invulling aangegeven, die al snel richting het genereren van refactoring guidance ging. Het heeft me ook de mogelijkheid gegeven om op basis van de brede verkenning een visie op te stellen voor de ‘ideale refactor guidance tool’. Waar ik wel lang voor mezelf in onduidelijkheid bleef, was waar mijn onderzoek nu precies een bijdrage aan leverde. De literatuurstudies en het vergelijken van mijn ideeën met de beperkte bestaande studies op dit gebied, hebben me uiteindelijk geholpen om mijn onderwerp een plek te geven in het geheel van bestaand werk. Ik heb me ook gedurende een groot deel van het traject te veel blindgestaard op het bouwen van het product. Het uiteindelijk besef dat er het idee ook generiek te beschrijven was in een model, was een mooi leermoment.

Een lastig punt in het onderzoeken blijft om het juiste materiaal in de grote hoeveelheid werk van bestaand onderzoeken te vinden. De verleiding is nog steeds groot om allerlei zijwegen in te slaan, met als uiteindelijk resultaat 100+ artikelen. Een tactiek was uiteindelijk om samenvatting, probleemstelling en conclusie door te nemen en de vraag te stellen of het relevant was voor mijn onderwerp. Al het andere werk werd direct geparkeerd, met het idee om het wellicht in de toekomst nog eens te bestuderen. Ik denk ook het feit dat het kernprobleem in het begin nog niet helder was, er voor gezorgd heeft dat het lastig is om de juiste zoektermen te bepalen en daarbij snel te bepalen of werk wel of niet een bijdrage aan het lopend onderzoek levert.

Als eindresultaat heb ik denk ik een mooie literatuurlijst dat refactoring vanuit veel verschillende hoeken heeft belicht en waarbij ik mijn toegepaste oplossingen in het onderzoek ook heb ik baseren op bestaand

werk. In de toekomst zou ik wel vanaf het begin meer selecteren op de herkomst van het werk. Het is uiteindelijke een kleine moeite om bijvoorbeeld peer reviewed journal artikelen van eenmalige conferentie publicaties te scheiden, maar dit was iets wat ik me pas later in het traject besefte. De kritische vraag kan nu gesteld worden of het materiaal wat gebruikt is wel altijd voldoende wetenschappelijke basis heeft. In sommige gevallen zou er voortgebouwd kunnen zijn op slechts 1 bron van beperkte kwaliteit.

Een terugblik op het mijn eerste opzet voor het uitvoeren van het onderzoek brengt me tot de conclusie dat deze te groot en te breed is opgezet. Ik wilde onderbouwde keuzes waarom ik zou focussen op bepaalde code smells. Ik wilde een tool die ik praktisch kon inzetten en deze tool moest uitgebreid geëvalueerd zijn met studenten. De insteek was in het begin vanuit een heel praktische toepassing, waarbij ik meer vanuit mijn engineering achtergrond met een product bezig was, dan me op dat ene specifieke vraagstuk te richten. Ik denk dat als ik me hier eerder van bewust geweest was. Ik meer tijd had kunnen spenderen aan het overbrengen van het idee en de scope vanaf de start een stuk beperkter had gehouden. In toekomstig onderzoek zou ik veel meer afbakenen waar de tijd naartoe gaat en het me niet laten verleiden tot het verspreiden van werk over wat eigenlijk meerdere onderzoeken zijn: theoretisch model, studenten ervaringen, code kwaliteit van studenten.

Aan mijn huidige onderzoeksopzet zou ik nu naast een betere afbakening van het onderwerp en vraagstuk in een vroeg stadium ook meer tijd hebben willen besteden aan een goede validatie van het model. Naar mijn idee is dit nu slechts beperkt gedaan omdat ik eigenlijk RAGs heb geconstrueerd en vanuit daar heb geredeneerd hoe het prototype in bepaalde contexten hieruit refactoring guidance zou moeten genereren. Dit genereren is gedaan op basis van voorgedefinieerde cases. Het zou veel interessanter zijn hoe het prototype zich gedraagt in een meer realistische situatie.

Afsluitend nog enkele reflecties op het schrijfproces en inhoud van mijn thesis. Het helder overbrengen van ideeën op schrift blijft een lastig proces waar ik me nu wel bewuster van ben tijdens het schrijfproces. Het werken met outlines was iets wat ik al wel deed, maar de ideeën gepresenteerd in het boek ‘the pyramid principle’ van Barbara Minto hebben me inzicht gegeven om beter te letten op de logische opbouw van deze outlines. Het heeft volgens mij veel bijgedragen aan de opbouw van de thesis.

Tekstueel zou ik nu het woord ‘we’ in de tekst heroverwegen. Lex heeft hier een opmerking over gemaakt en dat zette me aan het denken. Ik ben het woord gaan gebruiken op basis van artikel voorbeelden, waarin vaak meerdere auteurs betrokken zijn. Ik zie nu ook in dat het in de tekst op sommige punten verwarring kan geven over wie het nu eigenlijk gat. Daarnaast had ik ook nog graag mijn discussie en resultaten scherper neer willen zetten. Hier zit denk ik nog zeker ruimte voor verbeteringen. Ook hier geldt weer het principe van afbakening, wat neem ik wel en niet mee in de discussie. Het zijn de laatste hoofdstukken geweest, waar ik ten opzichte van de andere hoofdstukken weinig tijd voor heb gehad om het meerdere keren te herzien. Dit is denk ik direct ook een van de lastigste punten aan schrijven. Hoe tot een goed stuk werk te komen, waarbij niet de tekst te blijven herlezen en te blijven aanpassen? Misschien is het simpele antwoord, ervaring.

11 Appendix A – Refactoring Advice Graphs

This appendix contains the RAGs that have been constructed based on identified CCPs for *rename method* and *extract method*.

Rename method

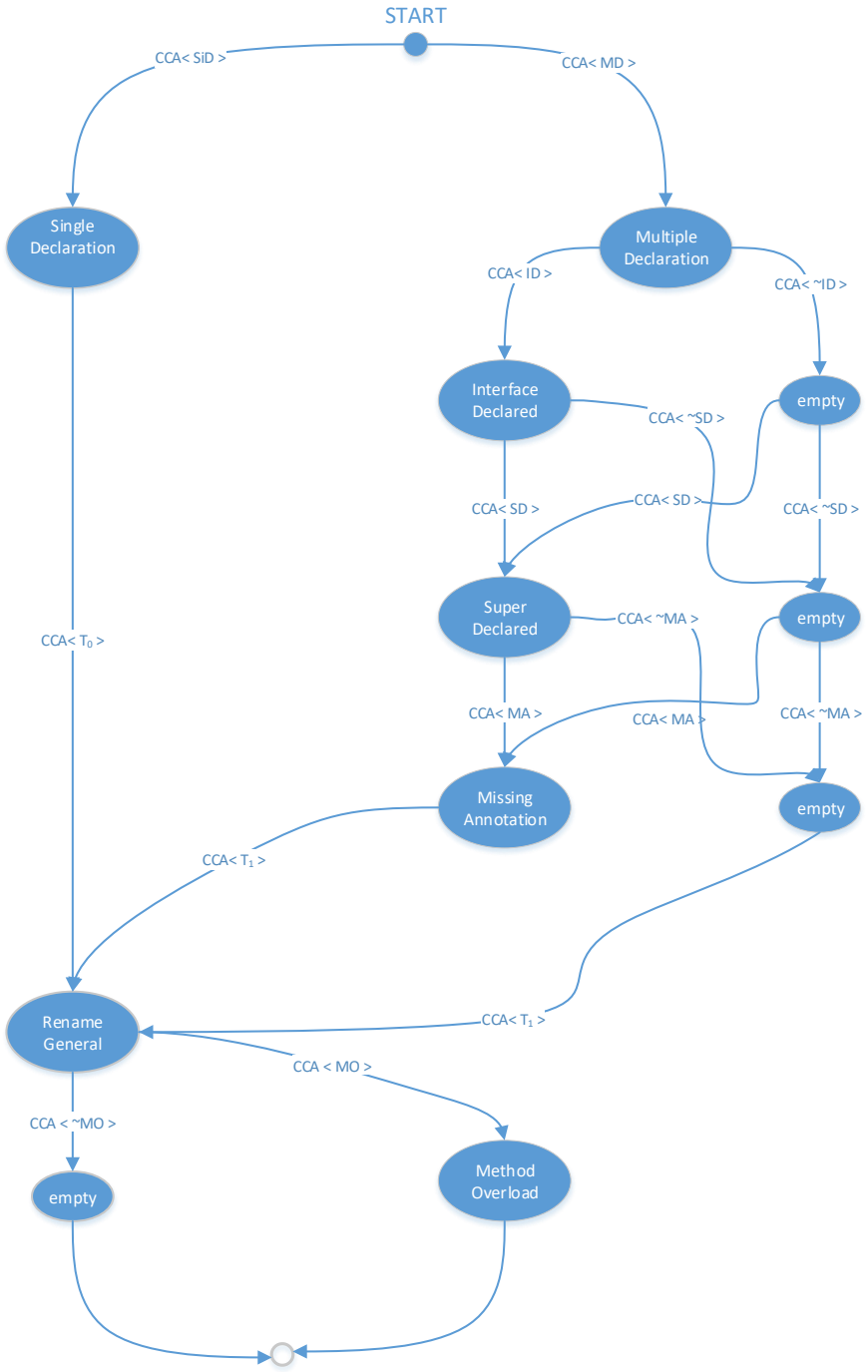


Figure 16 RAG - Rename Method

Extract method

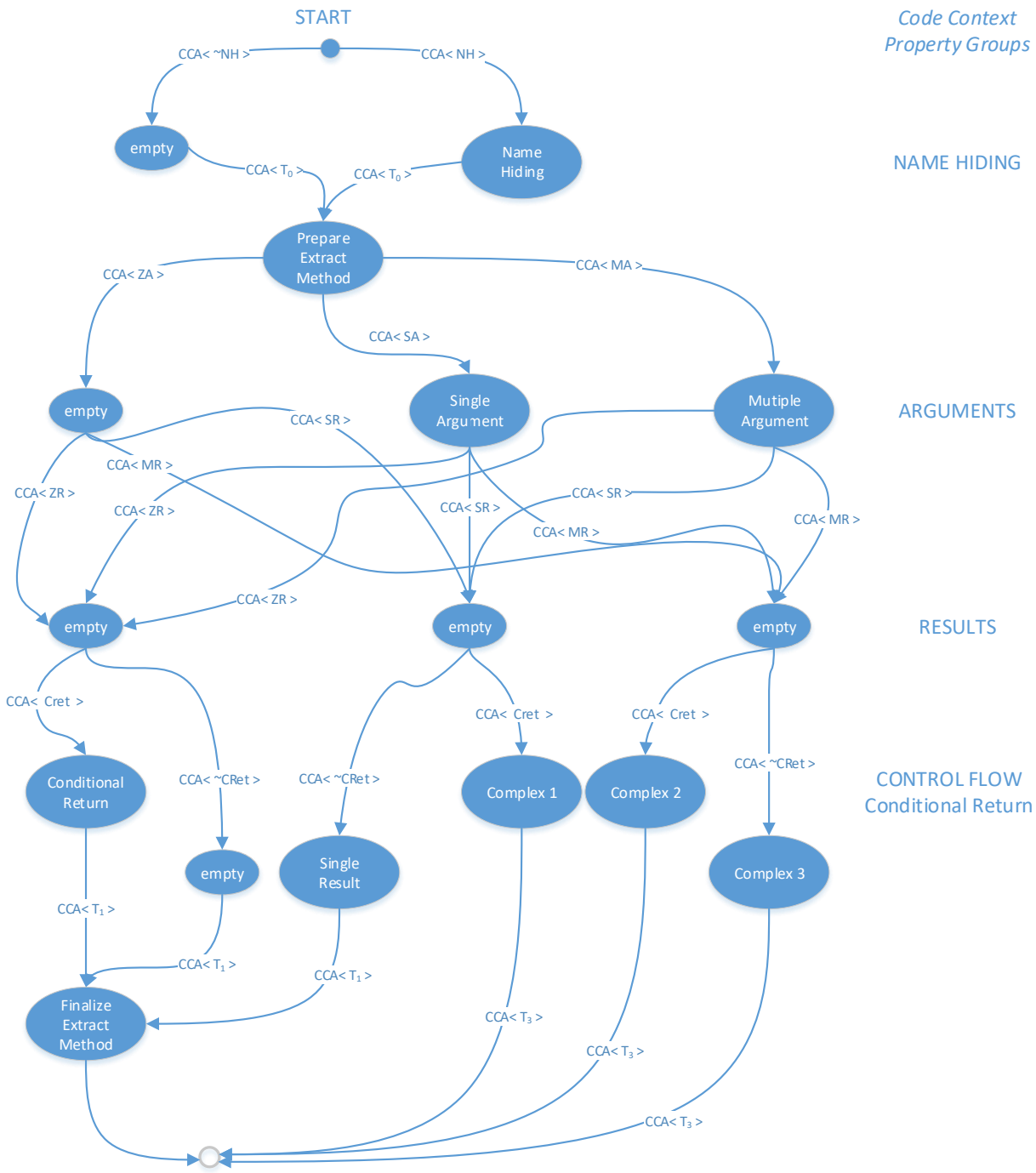


Figure 17 RAG - Extract Method

12 Appendix B – Identified Extract Method CCPs

This appendix gives a detailed description of the CCPs we have identified for extract method. For each of the CCPs we describe the meaning and how the context could be detected in code. These CCPs are used in the construction of the Extract Method RAG.

IDENTIFYING REFACTORING SCENARIOS

In the *extract method* mechanic described by Fowler [19] scenarios are distinguished based on how local declared variables in the source method (method from which code is extracted) are used by the code that will be extracted to the new target method.

We can distinguish three situations when moving a functional group of code from source method to target method. The extracted code in the target method:

1. Does not have any relation to local declared variables in the source method. [*DF No Dependencies*]
2. Reads one or more variables defined in the source method. [*DF Reads Local Defined Variables]*
3. Modifies one or more variables defined in the source method, which are used in the source method after the target method has been called. [*DF Modifies Live Local Defined Variables]*

What Fowler mechanics describe in his examples for case 2 and 3 is what later studies have been describing as preserving data flow [16, 49]. These studies extend the scenarios of extract method by also stating that name binding and control flow should be preserved when performing *extract method*.

Preserving data flow (DF)

Data flow is about assuring that the state of variables during their lifetime in the source method reflects the same state at any location in the source method after extract method has moved code to the target method. Wrong refactoring might lead to changes in behavior of variable state within the source method.

Preserving control flow (CF)

Control flow is the order in which individual statements, instructions or function calls are executed or evaluated. Wrong refactoring might lead to a break of control flow, which can lead to different behavior of the code. A simple example is given by a conditional return statement in the source method, that is moved to the target method in the *extract method* [*CF – Conditional Branching*].

Given the example below:

```
public int source(int b){

   if(b>10){
      b = 0;
      return b;}

   b++;
   return b;
}
```

Extracting the if-statement without preserving control flow, might result in code like below:

```
public int source(int b){
```

```
    b = resetWhenExceeds10(b);

    b++;
    return b;
}

public int resetWhenExceeds10(int b)
{
    if(b>10) {
        b = 0;
        return b;
    }
    else
        return b;
}
```

In the source method the end result of calling *source(11)* would lead to a result of 0. After extracting code to the target method we would now have in our example a final result of 12.

Within the Java version JDK8 we can distinguish three types of statements that can alter the control flow of executing statements [46]: *decision statements* (if, switch), *looping statements* (for, while, do-while) and *branching statements* (break, continue, return). Not mentioned here but also influencing control flow are exception handling statements. In our case we only worked out the conditional if-statement for our prototype. It would be interesting to investigate also the other statements that influence control flow in future work.

Preserving name binding (NB)

Name binding is how the compiler associates variables to a unique object in memory. Wrong refactoring can lead to changes in name bindings, which might go unnoticed. An example of this is when a variable in a method is hiding a property with the same name on class level [*NB – Local variable redefinition]*

Given the example below:

```
class A {

private int var = 0;

     public void print(string s){
         int var = 5;
        System.println(s + var);
     }
}
```

Extracting the formatting statement to a new target method without paying attention to name binding:

```
class A {

private int var = 0;

     public void print(string s){
         int var = 5;
        printFormatted(s);
     }

     public printFormatted(string s) {
        System.println(s + var);
     }
}
```

When calling original code with *print("Value: ")* will result in the output *Value: 5*.
After extraction the result would be *Value: 0.* This because we forgot to pass on the value of local variable *var*. The compiler will not notice because a variable with the same name exists as a property.

DF – No dependencies / Zero Arguments (ZA) AND Zero Results (ZR)

When there are no dependencies on any of the local defined variables in the original method. A straight on scenario can be followed to refactor the code.

The template advices has parametrized keywords for source method name and new target method name in their instructions, as also the name of the class in which the refactoring takes place.

DF –Reads 1 (Single Argument [SA]) or Multiple Local Defined Variables (Multiple Argument [MA]).

Within this context we would like to create awareness which local defined variables should be passed on to the new method that is extracted. Although the compiler will warn for any missing defined variables in the extracted code we think it still is useful to detect this context for following reasons:

1. With the presence of local defined variables that hide variables declared outside the original method, refactoring errors might be introduced easily. A hint could be generated to warn for this condition. See also the section about name hiding. (NH – Local variable hides class field)
2. When more than 1 variables should be passed on, hints can be generated if introduction of a parameter object is worth considering.
3. We can give feedback on which and how many variables should be passed on to the extracted method. In a possible future scenario where we want to track progress of a refactoring, we can determine if the right parameters are taken into account.

The template advice has parametrized keywords for the names of the local declared variables were the extracted method is depending on. Optionally, the source and target method could be included.

DF – Modifies 1 (Single Result [SR]) or Multiple Live Local Defined Variables (Multiple Result [MR])

Instead of modification of local variables, as defined by Fowler, we will be looking at the modification of variables that are live. This prevents that we return variables, that are not used in the original method after calling the method that is extracted.

A variable is live when it is assigned a value that is read later without being written again in between the write and the first read.

With the detection of this context we want to make the student aware which local defined variables in the original method will be changed in the extracted method and should be returned to the original method to preserve data flow. It is imaginable that a novice student might lose the overview when extracting from larger methods, where variables are not nicely grouped together.

In Java we can only return one value at a time. So there is a special case when more than one variable is live in the extracted code. In this case the solution to perform extract method becomes more complex. Multiple follow up refactorings can be considered at this moment. In our current solution we will present a list of possible approaches of which a student can choose. We let it for future work to further invest this scenario.

NB - Local variable redefinition (Variable hides class field [NH])

When moving methods it might prone to error that other variables are depending on when moving code around. Notify students when such a situation might be take place during refactoring.

Identified from discussions is name hiding

CF – Return in Conditional if-statement

More tricky is the return statement in a conditional statement. Extracting this type of code to a new method will break control flow and the compiler won't issue any errors.

For all of these branching statements the solution would be to return a Boolean from the extracted method that can be used to determine whether or not the branching statement in the original method should be executed.

13 Appendix C - CCPs Mapped to Advice Templates

This appendix gives an overview of identified CCPs and the template advices that are at this moment associated with their CCPD function. The overview only contains the implemented refactoring selectors currently in our prototype.

Rename method

Table 3 Defined Template Advices for each identified CCP

CCP	Advice
SD - Method declared only once	In the current context there is no risk in renaming method #method directly
SupD - Method overrides a method of one of the super classes	Method #method has been also defined in the following super classes:#class-list To eliminate any side-effects, I suggest to rename #method also to your new name in the following classes: #class-list
ID - Method declared in public interface	declaration exists in public interface #interface It is a good practice to: 1. Mark public #method deprecated in #interface with annotation '@Deprecated' 2. Add method with new name to interface #interface 3. Add method with new name in class #class 4. Cut content of #method in #class and paste into your new method 5. Place in old #method a direct return call to your newly created method. Example: return newName(); 6. Put above old #method @Depricated
MA - Method can have @Override annotation	You did not add @Override everywhere before the method definitions. We suggest to add @Override to each method listed below: #method-list
MO - Method overloaded in same class	There are methods present in your class hierarchy with the same name (method override), but different number of parameters. It is a good practice to also perform refactoring Rename Method also for these methods.
MD - Methods with same signature in class hierarchy	Method #method is not declared for the first time in class #class Here some advices follow:
default	Rename #method in class #class to your new name Build project. Resolve unresolved references to #method indicated by compiler by changing the old name to the new name. Run your automatic tests and solve issues.

Extract Method

Note here that the cases MR and SR cannot have a direct Advice related, but depend on the presence of CRet. We used Boolean description in the advices to distinguish the cases for MR and SR.

Table 4 Defined Template Advices for each identified CCP (Extract Method)

CCP	Advice
NH - Local variable hides class field	The code you extract contains variables *[#variable-list*] that are hiding fields in your class. Start by renaming your variables. This will prevent that variables in your extracted code will use accidentally class fields i.s.o. the local variable. The compiler will happily compile if you forget them.
ZA - Zero arguments must be passed to extracted code	-
SA - Single argument must be passed to extracted code	*V*ariable [#*argument-list*] is used in your new method and assigned a value before your method is called. Copy argument declaration for #*argument* as an input parameter into your new method.
MA - Multiple arguments must be passed to extracted code	Variables [#argument-list] are used in your new method and are assigned a value before this method is called. Copy argument declarations for variables *#argument-list* as new input parameters into your new method. Do these parameters have a close relation with each other? If so, consider grouping them in a new object that contains these parameters. This will reduce the number of needed input parameters for your new method.
ZR - Extracted code should return no result	-
SR - Extracted code should return single result	***(SR ^!CRet)*** Your extracted code contains variable [*#result-list*] that is needed later on in the calling method [*#method*]. Add return type to your new method that equals the type of variable [*#result-list*]. Return variable at end of your new method. In the original method [*#method*] assign the result after calling the new method to the local variable [*#result-list*] **(*SR ^CRet*)** Your extracted code contains a conditional return statement. Next to this fact 1 or more variable result need to be returned to the calling method [#method-name]. For these reasons there is not a straight resolution path to solve this refactoring. Several options exists to proceed[1]: a. Create a class which serves as a return object. All variables to be returned by your new method are hold in the return object. b. Eliminate the return statement. d. Can variables [#result-list] in [#method-name] be moved to a field of class [#class-name]? This would reduce the number of returned values from your extracted method.
MR - Extracted code should return multiple results	***(MR & CRet)*** Your extracted code contains variables [*#result-list*] which are needed later on in the original calling method [*#method*]. We can only return 1 result in the new method. For these reasons there is not a straight resolution path to solve this refactoring. ***(MR ^ !CRet)*** Your extracted code contains variables [*#result-list*] which are needed later on in the original calling method [*#method*]. We can only return 1 result in the new method. For these reasons there is not a straight resolution path to solve this refactoring. Several options exists to proceed[1] a. Create a class which serves as a return object. All variables to be returned by your new method are hold in the return object. b. Can variables [#result-list] in [#method-name] be moved to a field of class [#class-name]? This would reduce the number of returned values from your extracted method.

CRet - Conditional return present in extracted code	Your code contains a conditional return statement. Add Boolean result type to your new method description. Return true in the conditional expression. Return false at the end of your new method. Wrap your new method that is called in [$method-name] with "if(newmethod() == true) return;"
T_0 – default CCA, always evaluates to next vertex Advice: Prepare Extract Method	Create a new method where the extracted code should go to. Choose a name that covers what this extracted code actually does. Let the return type of your new method be *void* and let it have none parameters. Example: "void newName() { <Extracted code> }". Place a call to your new method just before the code that will be extracted from method [#method]. Let the code to extract untouched. We will delete it later. Copy the code to extract to the newly created method
T_1 – default Advice: Finalize Extract Method	Compile code (default) Solve any unresolved variable names in your newly created method by adding de declarations for these variables locally in the new method. Remove original code from [*#method*]. Remove variables that are not used (this is not supported yet). Run your automatic tests. If none make at least one for the new method.

15 Appendix D – Prototype Design

This appendix presents a quick overview of the generic design of the prototype. One important note to make is about refactoring selectors and decorators. This naming cannot be found back directly in the design.

Refactoring selector: In the prototype the logic to detect code context is in the *ContextDetector*. When the specific code context is detected, then the information that is relevant for the refactoring decorator is retrieved and stored in a *ParameterCollector* object. This is an object that holds all parameter definitions of all present *ContextDetectors*. There is in the prototype not a single decorator per selector.

```
public class MethodSingleDeclaration extends ContextDetector {

    private ClassMethodFinder _analyzer = null;
    private MethodDescriber _method = null;

    public MethodSingleDeclaration(ClassMethodFinder cmf, MethodDescriber method) {
        this._analyzer = cmf;
        this._method = method;
    }

    /**
     * Used in generic context builder
     * @param cc
     */
    public MethodSingleDeclaration(ContextConfiguration cc) {
        this._analyzer = cc.getCMFAnalyzer();
        this._method = cc.getMethodDescriber();
    }

    @Override
    public boolean detect() throws Exception {

        if(_analyzer != null)
        {
            if(!_analyzer.isMethodDefinedInSuperClass(_method) &&
                !_analyzer.isMethodDeclaredFirstTimeInInterface(_method))
            {
                getParameters().addSingleMethodName(this._method.fullTypeSignature());
                getParameters().addSingleClassName(this._analyzer.getQualifiedClassName());
            }
        }
        else
        {
            throw(new Exception("Analyzer = null"));
        }

        return !getParameters().getCollection().isEmpty();
    }

    @Override
    public CodeContext.CodeContextEnum getType() { return CodeContext.CodeContextEnum.MethodSingleDeclaration; }
}
```

Figure 18 An example of a ContextDetector which determines if a method has been declared only once. When detect() evaluates to true, parameters are added to the ParameterCollector object which is allocated in the base class.

Figure 19 gives an overview of the design of our prototype. The generator has been built upon the generic *ContextDetectorSetBuilder* and generic *ContextAnalyzer*. ProceduralGuidanceGenerator follows the generic process of:

1. SELECT appropriate AIG based on user input for refactoring to perform
2. SPECIFY relevant refactoring input parameters
3. CREATE & CONFIGURE Context detector set
4. DETECT code contexts
5. GENERATE refactoring procedures

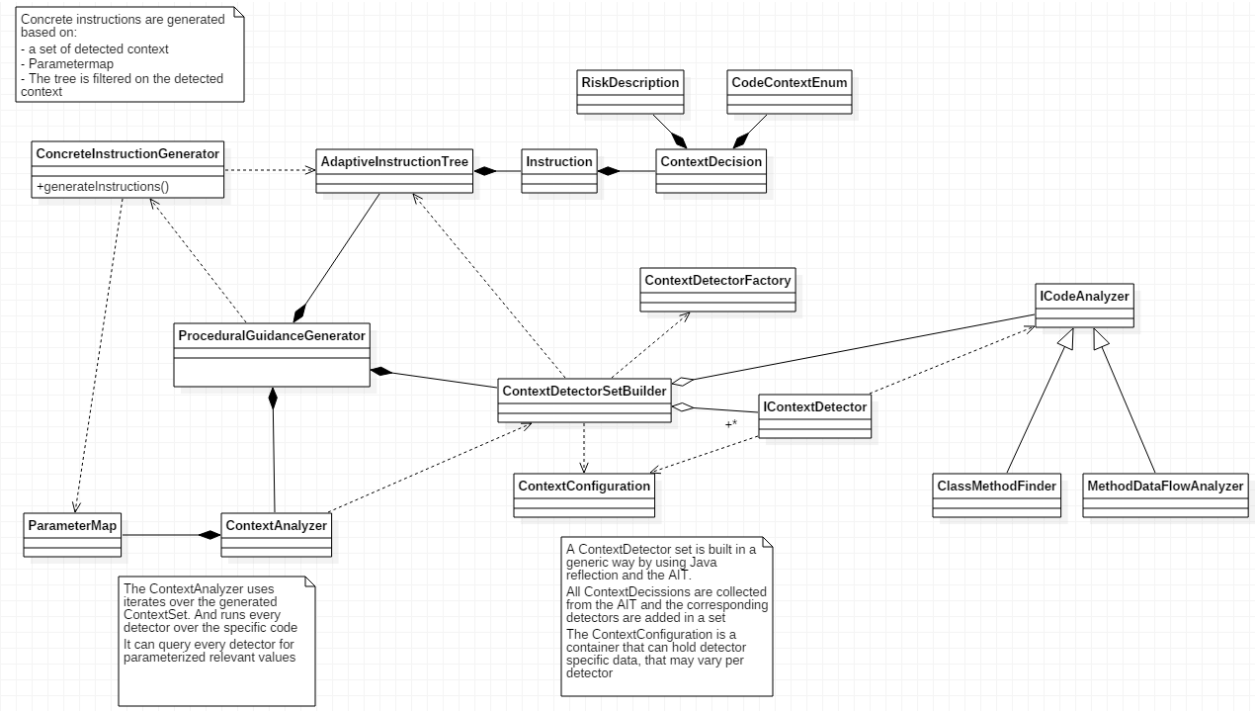


Figure 19 Design of the prototype.

Analysis of the code was done as described in earlier research on building an ITS (*Intelligent Tutoring System)* for learning to program Java [55]. They perform code analysis on an *abstract syntax tree* (AST). While we are depending in some cases on code context like e.g. determining presences of method signatures in inheritance trees, which cannot be determined by an AST. For this reason also symbol solving that is generated out of Java code. An AST is typically a tree representation of one Java file\class. This tree can be used to analyze code on a class scope, like e.g. analyzing exit criteria of a for-loop, determining intra method control flow, calculating complexity, etcetera. If we need information about the code in which other classes are involved, like e.g. determining the inheritance tree of a class for making a decision about the presence of overridden methods, then an AST is not sufficient. In these cases we need symbol tables that make it possible to resolve where types and variables are declared in other classes or files.

Several open source libraries are available that can generate AST and symbol tables. These libraries can also help us in navigating through the AST and retrieving the necessary information from AST or symbol tables. For our study we have been looking into the possibilities of the following libraries: ANTLR, Rascal, JavaParser and Spoon. The authors of Rascal and JavaParser confirmed that the tools could offer the both the AST and symbol resolving features for Java. Both of the tools are being actively developed and offer quick and good support by their developers. JavaParser uses a procedural programming paradigm in contrary with Rascal which is based on the functional programming paradigm. While we are more familiar with the procedural programming paradigm we decided to build our refactoring selectors based upon functionality offered by JavaParser 0.6.0.

16 Appendix E – Example of a Generated Refactoring Guidance

We present here the automatically generated *rename method* refactoring guidance that was the results of a rename of method *GetAccountName*() in class API_SpecialImplementation, located at line 33. Declared in *API_Rename.java*.

For readability we have included the complete generated text below the Figure 2.

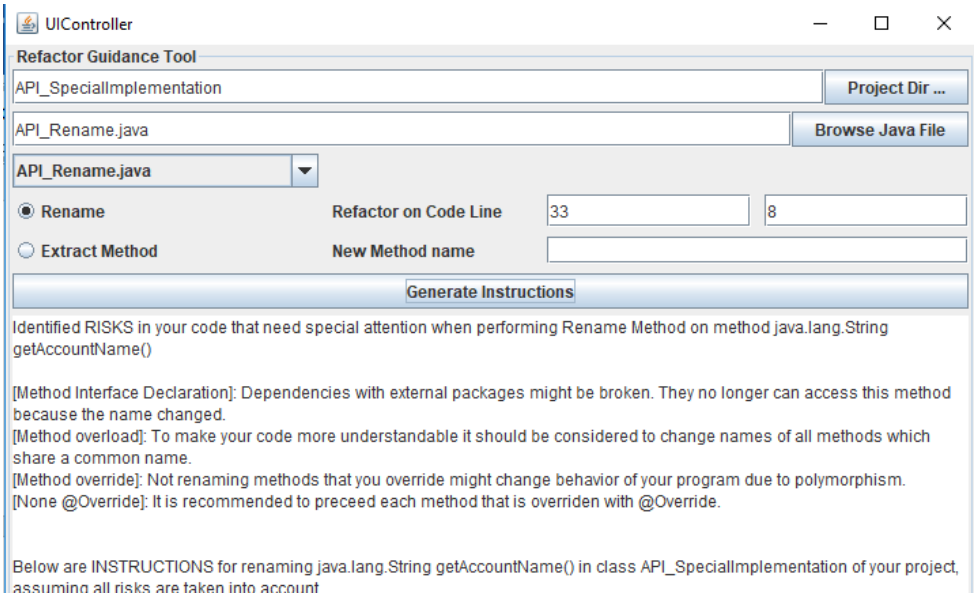


Figure 2 Screenshot of prototype to generate refactoring procedure

---

Identified **RISKS** in your code that need special attention when performing Rename Method on method java.lang.String getAccountName()

[Method Interface Declaration]: Dependencies with external packages might be broken. They no longer can access this method because the name changed.

[Method overload]: To make your code more understandable it should be considered to change names of all methods which share a common name.

[Method override]: Not renaming methods that you override might change behavior of your program due to polymorphism.

[None @Override]: It is recommended to precede each method that is overridden with @Override.

Below are **INSTRUCTIONS** for renaming java.lang.String getAccountName() in class API_SpecialImplementation of your project, assuming all risks are taken into account.

Feel free to skip specific steps for risks which you think are not relevant. Steps are marked with [] for specific risks.

Method java.lang.String getAccountName() is not declared for the first time in class API_SpecialImplementation .

[Method Interface Declaration]

A declaration exists in (public) interface API_Interface.

It is a good practice to

1. Mark public java.lang.String getAccountName() deprecated in API_Interface with annotation '@Deprecated'

2. Add method with new name to interface API_Interface

3. Add method with new name in class API_SpecialImplementation

4. Cut content of java.lang.String getAccountName() in API_SpecialImplementation and paste into your new method.

5. Place in java.lang.String getAccountName() a direct return call to your newly created method. Example: return newName();)

[Method Override]

Method java.lang.String getAccountName() has been defined in the following superclasses:

API_Implementation

To eliminate any side-effect risks, I suggest to rename java.lang.String getAccountName() also to your new name in:

APIImplementation

[None @Override]

For the listed methods @Override has not been added everywhere. Before renaming any methods, Add @Override above methods

API_SpecialImplementation :: java.lang.String getAccountName(), API_Implementation :: java.lang.String getAccountName()

Rename java.lang.String getAccountName() in class API_SpecialImplementation to your new name

Build project.

Resolve unresolved references to java.lang.String getAccountName() indicated by compiler by changing the old name to the new name.

Run your automatic tests and solve building issues.

[Method Overload]

There are methods present in your class hierarchy with the same name (method override), but different number of parameters.

It is a good practice to also perform refactoring Rename Method also for these methods.

17 Appendix F – Setup Semi-Structured Interviews

In this appendix we shortly summarize the set-up and questions asked in the semi-structured interviews we had when evaluating our concept and generated refactoring guidance with students

Set-up after iteration two

We performed a semi-structured interview with four individual students after the first iteration of our study. The iteration resulted in a prototype that could instantiate refactoring guidance for rename method refactoring. We had a 30 minute interview in which we started with some general questions on how a student would solve a certain rename action without the tool. This to get some impression on how students would normally perform a refactoring.

Questions

1. Wat versta je onder refactoring?
2. Kun je jou strategie uitleggen hoe je een rename actie uit zou voeren? (manueel/automatisch).
3. Gebruik je wel eens tools voor refactoring? Ja, welke
4. Hoe zeker ben je dat jou wijziging geen side-effects heeft op de overige code?

After these questions we introduced the refactoring guidance tool and explained its purpose. The textual output of the tool was shown and several questions were asked to give feedback on how they perceived the tool and where they think improvements might be needed.

Set-up after iteration three

The third iteration was concluded with a semi-structured group interview with two groups of students which consist each of five third year undergraduate SE students. We used our prototype to instantiate refactoring guidance for rename method and extract method. These were the same refactoring guidance as instantiated in chapter 16. The instantiated refactoring guidance was printed and handed out to each individual student. The students also received the original code that could be loaded and compiled in their IDE. It was asked to turn of code suggestions from the IDE, so that hints from the IDE could not be a distraction in the refactoring process they had to manually perform. The students were given 15 minutes to work out the refactoring instructions, by manually performing the refactoring in the handed out Java projects. Any ambiguity in the instantiated refactoring guidance could be noted on the handed out paper.

The group was asked after the 15 minutes if they had been able to perform the task and if there were unclear parts in the instructions. The group discussion was continued by asking prepared questions.

Questions asked: (italic are notes of answers given)

1. Vraag de student om op basis van instructie uit de tool, de rename refactoring uit te voeren.
2. Zou een gegenereerde stappenplan als deze potentieel kunnen helpen om je een betere keuze te laten maken hoe een refactoring uit te voeren, dan als je deze informatie niet zou hebben?
3. (ja) Wat helpt je dan in dit specifieke geval?
4. (nee) Wat heb je meer aan informatie nodig?
5. Heb je nu een beter begrip van het probleem wat hier aangestipt wordt?

6. Kun je me vertellen waar in de instructies je aanpassingen zou willen?

1. Zijn er punten die nieuw waren?
2. Helpt een dergelijk stappenplan om beter inzicht te krijgen in hoe je gestructureerd een refactoring kan aanpakken.
   SonarCube geeft de code smells.
   Risico + Aanpassingen is sterk.
   Automatische tool: pas als het faalt dan is het zichtbaar. Je vertrouwt op de tool. Andere tools werken anders.
3. Zijn er andere voordelen die jullie voorzien voor beginnende 2e jaars studenten, als je via een dergelijke stappenplan enkele opdrachten uitvoert.
   Meer inzicht in je code. Meer op software engineering ipv alleen focus op syntax. Risico's. Je leert meer conceptueel
4. Zie je ook toepasbaarheid van een dergelijke tool in je eigen projecten.
   Validatie, niewe ideeën opdoen. Minder dan in s2.
5. Wat zijn onduidelijkheden in de tool?
6. Wat zijn andere mogelijke verbeteringen in de tool?
7. Op een schaal van 1..10; hoe nuttig is een dergelijk tool?
8. Wat zouden jullie toevoegen aan een deze tool?
   Zelf detectors kunnen toevoegen door studenten.
   Tool checked of je de suggesties juist opvolgt.
   IISD in een proftaak verder laten uitontwikkelen.
9. Zijn er punten die nieuw waren?
10. Helpt een dergelijk stappenplan om beter inzicht te krijgen in hoe je gestructureerd een refactoring kan aanpakken.
    Zeker zinvol, maar kijk uit met te hersendood natikken
11. Zijn er andere voordelen die jullie voorzien voor beginnende 2e jaars studenten, als je via een dergelijke stappenplan enkele opdrachten uitvoert.
    Beter idee van de structuur en hoe je moet coderen
12. Zie je ook toepasbaarheid van een dergelijke tool in je eigen projecten.

13. Wat zijn onduidelijkheden in de tool?
    Geef van tevoren het doel aan waar je naartoe werkt; spelfouten; leesbaarheid

14. Wat zijn andere mogelijke verbeteringen in de tool?

15. Op een schaal van 1..10; hoe nuttig is een dergelijk tool?

18 Appendix K – Code Smells Quick-Scan of Java projects Written by Undergraduate SE Students

We prefer to investigate those refactorings in the context of our research that would give undergraduate SE students most benefits when they could be provided with instructions on how to manually perform a certain refactoring. For this we have been looking for studies that have been analyzing undergraduate SE student projects for common code smells or code refactorings that are often performed.

As presented in the introduction a few studies have been performed looking at student's software quality from a code metrics perspective or coding style (references). To the best of our knowledge there have been no studies that have analyzed student software projects with respect to code refactoring behavior or code smells.

For this reason we decided to assess a small set of 3rd year undergraduate SE Java projects to get a first impression what are common code smells in software of students that almost finished their study. This was done by performing a quick-scan for presence of code smells in these projects. Based on this inventory and some cues in other literature sources we have defined a top 5 of refactorings that seem likely candidates to solve those smells that were identified in our students' projects datasets.

Inventory SETUP

CHARACTERISTICS STUDENT GROUP

We have gathered completed software projects from one of our semester 6 - February 2017 Fontys ICT – Software Engineering classes. These students are in their last semester where software courses from the regular software engineering curriculum are presented. After this semester students will either run their internship graduation or they will follow a minor first before their starting their graduation internship.

Every student in this group has at least 2,5 years of experience in programming C# and Java projects in an educational setting. In every prior semester of their education they have been building a large software project in a joint group effort. The content of these software projects are often formulated by one of our partners in education [6]. Next to these group projects every students has to apply their knowledge in small individual defined software projects. Both type of projects are used to demonstrate the capability of applying software engineering concepts in realistic software cases. All students have also at least a half year of professional work experience in a software project during an internship at a software company.

The students involved in this project were familiar with the concept of refactoring and code smells, but did not have formal training on these topic via a dedicated course.

USED DATASET

Our used dataset is based on Java projects submitted end of semester 6 which were provided by a link to personal and group GIT archives of our students. Snapshots of these projects were downloaded as zip files from GIT. Projects were all written in Java 8 EE in combination with front end frameworks like e.g. AngularJS.

Test cases were removed from the datasets. We instruct our students to keep test methods small, testing only single or few pieces of functionality in one test case. Logic is limited in these test classes, so

is the dependency from this test cases to other classes and methods. For this reasons we decided that analyzing for code smells in this test code would probably not reveal many code smells.

All projects were tested if they could be compiled. In a few cases additional libraries were needed to make compilation possible. Projects that we couldn't compile by adding additional libraries were excluded from the dataset. This resulted in a set of 18 software workspaces available for our assessment.

Empty projects or code projects that had were not Java based were removed from the remaining 18 software workspaces.

The end result of this preprocessing steps resulted in 11 Java projects that were suitable for the inventory of code smells.

COUNTING CODE SMELLS

We decided for an automated detection of code smells. Compared to manual analysis we think that automated analysis will present a more uniform result in a time efficient way. This choice does however limit us in not being able to detect all possible code smells.

A previous study placed the 22 code smells of Fowler in a taxonomy and for each of them it was determined if automatic detection is likely to be feasible or not *[37].* We removed code smells that were marked as being hard to detect. We also removed those code smells that were placed in the taxonomy of *change preventer*. Change preventers are those code smells where change is needed in many places when you make a change in one location of the code. Typically detection of this type of problem is done by looking at the evolution of code in a software repository. This type of data we do not have available for our chosen target group. We ended up with 11 code smells of which the idea is that they can be automatically detected: Duplicate code, Long method, Large class, Long parameter list, Feature Envy, Data Clumps, Primitive obsession, Speculative generality, Message chains, Middle man and Data classes.

Available tools mentioned in literature that are able to detect one or more of the last mentioned code smells are: PMD, JDeodorant, JSpirit, Stench Blossom, JSmell and CodeNose. Stench Blossom, JSmell and CodeNose were no longer publicly available or could not be configured correctly.

An overview of smells, detection and tools can be found in Figure 20.

With the remaining tools we were able to automatically detect the following code smells: Duplicate methods, Long methods, Large classes, Long parameter list, Feature envy and Data class. The result of detecting code smells in our student project can be found in

We have counted the occurrences of smells indicated by each tool. Based on these outcomes we came to the following most occurring code smells in our undergraduate SE student code:

1. Long Method
2. Long Parameter List
3. Feature Envy
4. Large class

FROM CODE SMELLS TO REFACTORING PROCEDURES

Fowler describes which refactoring procedures are candidates for solving code smells [19]. We have put refactoring procedures and code smells in a matrix as depicted in

We counted for each code smell of our top 4 which refactorings where used most. This showed that for our code smells *Extract method* and *Extract Class* are most relevant refactorings.

We compared this with which code refactorings occur 3 or more times when summing up all refactorings in our matrix. This lead to the following refactorings in most occurring order: Move method, Move Field, Extract Method, Inline Class, and Introduce Parameter Object. In interesting fact that shows up is that with these 6 refactorings are involved in 14 out of 22 refactorings.

Comparing these two lists of top refactorings makes *Extract method* or *Extract class* top candidates to further investigate in our research. Although *extract class* is used in more refactorings we choose to finally put *extract method* on top position, because this refactoring seems to be more relevant to our students based on the identified code smells.

A PRIORITIZED SET OF REFACTORINGS

- Extract Method
- Refactoring procedure shared by 3 out of 5 in student's code smells
- In top 5 of most used refactoring procedures looked at all refactorings
- Extract Class
- Move Method
- Move Field
- Extract Interface

Several arguments can be formulated that poses threads to the validity of this analysis:
1. The automatic tools we have used are not able to detect all of the 22 code smells.

2. Studies show us that tools to automatically detect code smells result in many false negatives. We tried to compensate for this to run multiple tools which all have their own strength, algorithm.

3. We only analyzed code of one class of one specific educational institute

4. Primary motivation of students might not always be to write high quality code if not asked for it

Still the results we could conclude from this dataset did match our expectations..

For more information on how the prioritized list was made, contact the author

Also the article of (Wilking, 2007) confirms the selection we have made in our study. In this article it is stated that rename method and extract method are one of the most common refactorings applied.

Explain that *rename method* is probably not that strange to take up. In practical setting I used it quite often to indicate to students that the meaning of the method is unclear

As we have seen in the analysis of our student code, extracting a method from a piece of code is a refactoring that probably is one of the most used refactorings for a student when improving his code. This is in line with the statement that refactoring of long methods is one of the most used refactoring activities (Fowler, 1999).

The outcomes are not that different from what has been identified in literature in different context than education. (Extract method often mentioned as one of the most important)

The table shows the results of which tools in existing work can be used to detect a specific code smell.

Code Smell	Description	Metrics	Tools
Duplicated code	Repeating the same code through the code		PMD,JDeodorant
Long Method	The longer a method, the bigger the chance that it holds multiple functionalities and relates to many external dependencies	Length of method Cohesion to outside complexity [1]	JDeodorant
Large Class	A class has clear more than one functionality. A.k.a. god class	Number of fields	JDeodorant,
Long parameter list	Passing in long number of parameters	Count number of parameters [2,3]	PMD
Divergent Change	Making a change at one clear point in the code, has always the effect of change in other parts of the code	*Change preventer, this needs archive analysis*	-
Shotgun Surgery	To make a change in code, it is necessary to make changes in many different classes.	*Change preventer, this needs archive analyses*	-
Feature Envy	A method is using more data of other classes that its own internal data. It seems to like the other class more.		Jdeodorant
Data Clumps	The same data items appear everywhere together (fields, method params).		Stench Blossom [4]
Primitive Obsession	Logically related primitives are not grouped into a class/structure	*Hard to measure*[5]	JSmell
Switch statements	Too much usage of switches in code, where polymorphic solutions could be used	*Hard to measure*[5]	Not so reliable, but we could count the occurences of switches and analyse manually if they are a switch smell.
Parallel Inheritance Hierarchies	Two or more class hierarchies that need to undergo adaptation when one of the hierarchies changes	Needs archive analysis[6]	-
Lazy Class	Classes that have no functional reason to exist.	No definition found for detection	
Speculative generality	Classes that are there for 'future' purposes, but are not used at all or only by test cases.	Find dead code	JSmell
Temporary Field	A field in an object is only used in very specific circumstances, by a small part of the methods in the object. Isn't it actually candidate for an object on its won?	*Hard to measure*[5]	
Message Chains	An object is requested, on which an object is requested, etcetera. A chain of object calls is created.	*Hard to measure*[5]	JSmell
Middle Man	The only functionality of a class is delegating tasks to other classes.	*No field declarations*	CodeNose[7]
Inappropriate Intimacy	Classes who have too much dependencies on (private) internal fields	Coupling to datafields of other objects	
Alternative classes with different Interfaces	Two or more classes exist that actually offer the same behavior, but hidden behind a different interface.	*Hard to measure*[5]	
Incomplete library Classes	Libraries that do not offer desired functionality anymore, so workarounds are created in classes that use them.	*Hard to measure*[5]	
Data Classes	Classes with only data-fields and no functionality.	Cyclic complexit, nr. Of methods, number of fields	JSmell

Refused Bequest	Children that do not use the parent's original methods at all or give methods a complete new behavior not related to the earlier functionality. Also unimplemented interface methods indicate this smell.	*Hard to measure*[5]...	
Comments	Comments might be an indication that naming is not chosen well for methods, or that there is actual code in that might be moved to another location "The best comment is a good name for a method or class"	*Hard to measure*[5]	

[1] charalampidou_cohesion_long_method.pdf

[2] Fowler

[3] https://www.javaworld.com/article/2074962/core-java/too-many-parameters-in-java-methods-part-8-tooling.html

[4] Automatic detection of bad smells in code – an experimental assessment

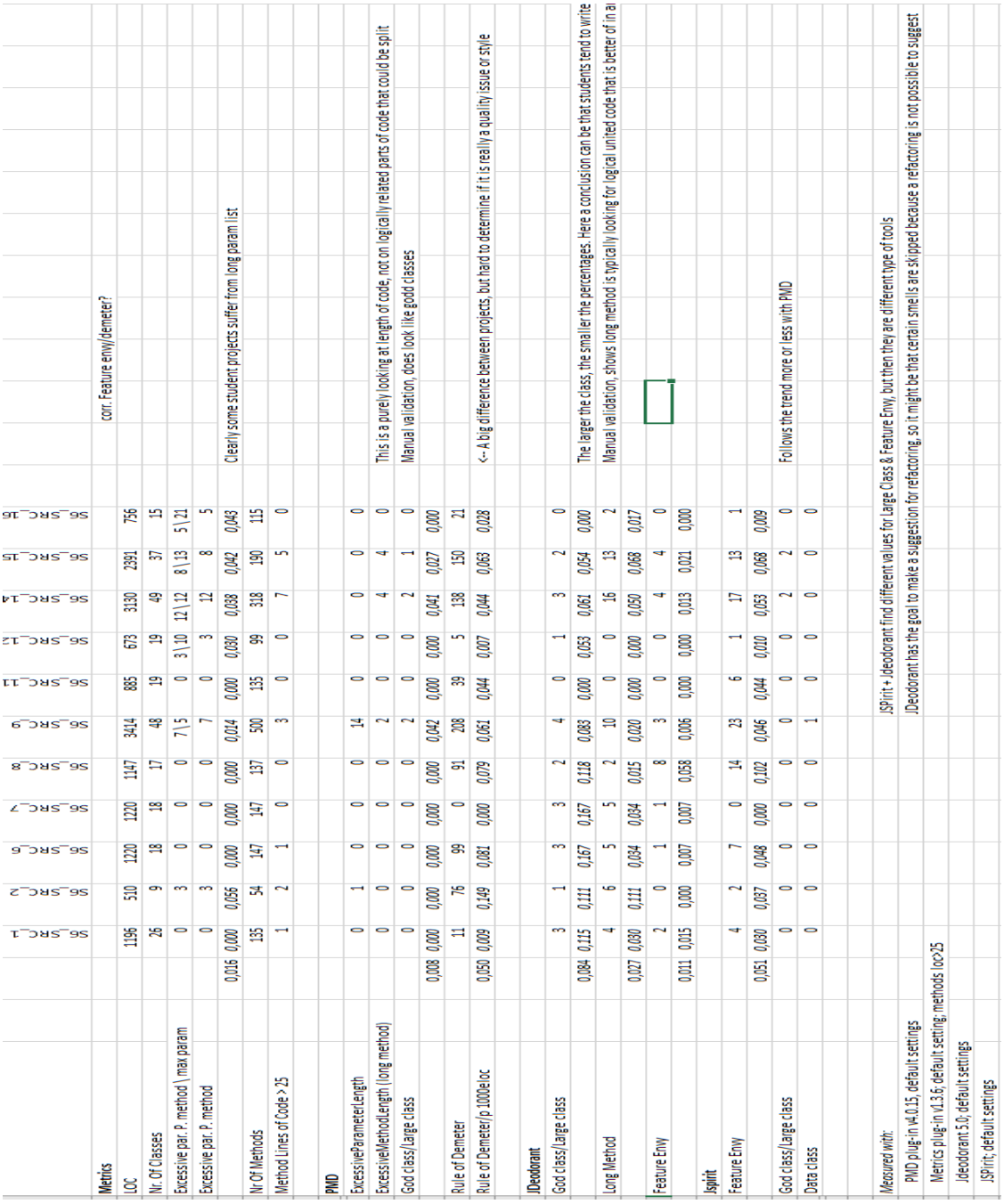
[5] Mantyla thesis, Bad code smells – Taxonomy & measures

[6] Identifying entities that change, Girba

[7] Code smell detection in Eclipse, S. Slinger

[8] https://www.slideshare.net/jimbethancourt/refactor-to-the-limit

This table gives an overview of identified code smells in the software projects made by 3rd year undergraduate SE students.
Some additionally notes were added for future analysis.



Metrics																corr. Feature envy/demeter?
LOC	1196	510	1220	1220	1147	3414	885	673	3130	2391	756					
Nr. Of Classes	26	9	18	18	17	48	19	19	49	37	15					
Excessive par. P. method \ max param	0	3	0	0	0	7 \ 5	0	3 \ 10	12 \ 12	8 \ 13	5 \ 21					
Excessive par. P. method	0	3	0	0	0	7	0	3	12	8	5					
	0,016	0,000	0,066	0,000	0,000	0,014	0,000	0,030	0,038	0,042	0,043					Clearly some student projects suffer from long param list
Nr Of Methods	135	54	147	147	137	500	135	99	318	190	115					
Method Lines of Code > 25	1	2	1	0	0	3	0	0	7	5	0					
PMD																
ExcessiveParameterLength	0	1	0	0	0	14	0	0	0	0	0					
ExcessiveMethodLength (long method)	0	0	0	0	0	2	0	0	4	4	0					This is a purely looking at length of code, not on logically related parts of code that could be split
God class/Large class	0	0	0	0	0	2	0	0	2	1	0					Manual validation, does look like godd classes
	0,008	0,000	0,000	0,000	0,000	0,042	0,000	0,000	0,041	0,027	0,000					
Rule of Demeter	11	76	99	0	91	208	39	5	138	150	21					
Rule of Demeter/p 1000eloc	0,050	0,009	0,149	0,081	0,000	0,079	0,061	0,044	0,007	0,044	0,063	0,063	0,028			<-- A big difference between projects, but hard to determine iif it is really a quality issue or style
JDeodorant																
God class/Large class	3	1	3	3	2	4	0	1	3	2	0					
	0,084	0,115	0,111	0,167	0,118	0,083	0,000	0,033	0,061	0,054	0,000					The larger the class, the smaller the percentages. Here a conclusion can be that students tend to write
Long Method	4	6	5	5	2	10	0	0	16	13	2					Manual validation, shows long method is typically looking for logical united code that is better of in a
	0,027	0,030	0,111	0,034	0,034	0,015	0,020	0,000	0,000	0,050	0,068	0,017				
Feature Envy	2	0	1	1	8	3	0	0	4	4	0					
	0,011	0,015	0,000	0,007	0,007	0,058	0,006	0,000	0,000	0,013	0,021	0,000				
Jspirit																
Feature Envy	4	2	7	0	14	23	6	1	17	13	1					
	0,051	0,080	0,037	0,048	0,000	0,102	0,046	0,044	0,010	0,053	0,068	0,009				
God class/Large class	0	0	0	0	0	0	0	0	2	2	0					Follows the trend more or less with PMD
Data class	0	0	0	0	0	1	0	0	0	0	0					
Measured with:																JSPirit + JDeodorant Find different values for Large Class & Feature Envy, but then they are different type of tools
PMD plug-in v4.0.15; default settings																JDeodorant has the goal to make a suggestion for refactoring, so it might be that certain smells are skipped because a refactoring is not possible to suggest
Metrics plug-in v1.3.6; default setting; methods loc>25																
Jdeodorant 5.0; default settings																
JSPirit; default settings																

Figure 20 Overview analysis student projects

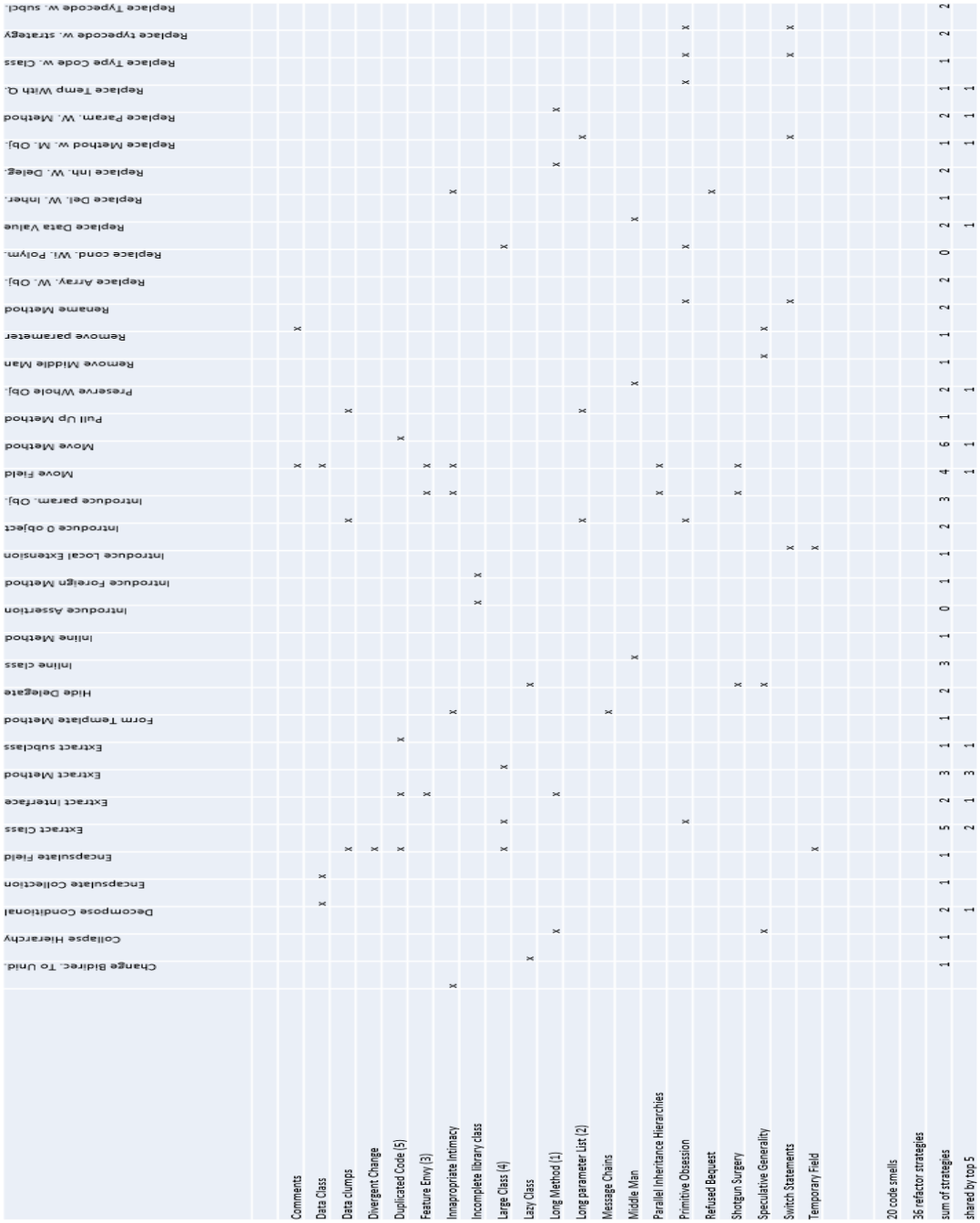


Figure 21 Code smells related to Refactoring Procedures