MASTER'S THESIS

Contextual Refactoring

Towards Risk-Driven Fowler based Refactoring Guidance

Hilberink, H.

Award date:
2021

Contextual Refactoring

Towards Risk-Driven Fowler based Refactoring Guidance

Herman Hilberink

Student:
Date: 31/10/2021

Open Universiteit
www.ou.nl

Contextual Refactoring

Towards Risk-Driven Fowler based Refactoring Guidance

by

Herman Hilberink

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Science
Master Software Engineering
to be defended publicly on Thursday November 11, 2021 at 11:11 AM.

Date:	October 31, 2021	
Student number:		
Course code:	IM9906	
Thesis committee:	Dr. ir. S. (Sylvia) Stuurman (chair),	Open University
	Dr. ir. H. J. M. (Harrie) Passier (1st supervisor),	Open University
	Prof. dr. A. (Lex) Bijlsma (2nd supervisor),	Open University

Open Universiteit
www.ou.nl

Acknowledgments

This research project would not have been possible without the ability to create. I could not do it alone; as such, a tribute to everyone who contributed or who has taught me how to do.

The two most important women, my partner for life, Rosé, and my dear mother, earn the most profound respect and love for supporting me through times that sometimes felt have a roller coaster ride of mixed emotions and ups and downs. The people I care about, my family and best friends, stood by my side all along. When time went by, even before starting this journey to do this master study, Rosé kept me motivated to pursue my dreams.

The attitude of never giving up, undoubtedly inherited from my father, my ever best friend, is what kept me going. My dad was my anchoring point for persisting in the job. Sadly he passed away recently during this graduation assignment. Unfortunately, my caring parents and stepmother lately needed to have cared for themselves.

The experience of being a student again, besides working as an IT professional, took me a considerable lot of energy and had been giving me enormous amounts of knowledge and satisfaction in return.

Special thanks go out to my fellow researchers, Evert Verduin and William Wernsen. As 'Refactoring Buddies' for life, we participated in numerous online Saturday morning sessions, which led to an explosion of mutually reinforcing knowledge gains. I hope (and secretly expect) that the necessary sessions will continue to take our concepts and product realizations to a higher level.

Lest we forget, I want to express my utmost appreciation to my supervisors, dr. ir. Harrie Passier and prof. dr. Lex Bijlsma. Without them, I could not have achieved as I stand today. They assisted me when I got stuck, especially during the writing period. Their input and support are invaluable!

Herman Hilberink
Almelo, october 2021

Contents

List of Figures

List of Tables

Summary

This research mainly focuses on software refactoring. By refactoring source code, we achieve higher quality and better maintainability of the software.

LEARNING REFACTORING Software refactoring is a typical activity of software developers. One of the requirements of successful refactoring is that one has to be proficient at performing refactoring. This is because refactoring is a complex and challenging activity. How to refactor can differ per situation, state of the code, and the objectives. Learning these skills and acquire knowledge is not easy and requires much practice.

Many of the refactoring techniques and methods are described in literature. We mainly focus on Fowler's catalog of refactorings from his standard work *Refactoring: Improving the Design of Existing Code*. He emphasizes the most essential quality, that of "behavior preservation", the preservation of expected functionality.

RECOGNIZE HAZARDS Recognizing "behavior preservation breaks" is to learn by pointing out the dangers that loom during refactoring. The method we use to provide insight is by already pointing out the potential dangers before refactoring.

GUIDANCE BASED ADVICE Specifically in this research, we therefore focus on the educational aspect of refactoring, which we refer to as 'Procedural Refactoring Guidance'. Students are assisted by solid advice, based on the selected refactoring and the state of the code. The main goal is to learn by doing and try to avoid making mistakes next time.

SUPPORT THROUGH TOOLING In order to facilitate guidance, it is crucial to develop the necessary tooling for this.Our main effort, described in this research report, is to develop a framework to support risk-based refactoring guidance. This Framework provides an elaboration of the topics below.

CONCEPT OF DETECTOR CHAINING To investigate the state of the code, we use the principle of performing source code diagnostics. The investigation is made possible by a chain of detectors, that each look for certain aspects of the code. The interpretation of the findings found is not the responsibility of the detector itself, but outsourced.

VERDICT IDEA For identifying potential danger, we introduce a new approach to reasoning about the state of the current code concerning the steps to take prescribed refactoring. The result of this reasoning process is a statement. Based on this ruling, the tooling formulates advice to the student.

FOWLER REFACTORING DECOMPOSITION Any refactoring that we (will) support must be reduced to elementary Microsteps of code changes. This Microsteps form the basis for risk assessment and its dependent advice. Because Fowler describes his refactoring procedures at a coarse grained level, the process has been devised how to derive these fine grained Microsteps.

SCENARIO BASED ADVICE The Microsteps to be performed affect the state of the code by making changes to the code. A mechanism is needed to develop the scenarios that can map the possible changes and the danger they pose.

WORKING EXAMPLES Testing our framework is demonstrated by a prototype tool with working examples.

Samenvatting

Dit onderzoek richt zich hoofdzakelijk op het gebied van software refactoring. Met behulp van het refactoren van source code bewerkstelligen we een hogere kwaliteit en betere onderhoudbaarheid van de software.

HET AANLEREN VAN REFACTORING Software refactoring is een typische activiteit van software ontwikkelaars. Een van de vereisten van een geslaagde refactoring is dat men bedreven moet zijn in het uitvoeren van een refactoring. Dit komt omdat refactoring een moeilijke en complexe activiteit is. Het aanleren van deze vaardigheden en het verwerven van kennis gaat niet vanzelf en vergt veel oefening. Per situatie, toestand van de code en de doelstellingen kan de manier hoe te refactoren verschillend zijn.

Veel van de refactoring technieken en methodes staan beschreven in literatuur. Wij focussen ons voornamelijk op de catalogus van Refactorings van Fowler uit zijn standaard werk *Refactoring: Improving the Design of Existing Code*. Hij hamert daarbij op de meest belangrijke eigenschap, dat van "behavior preservation", het behoud van verwacht gedrag.

GEVAREN HERKENNEN Het herkennen van "behavior preservation breaks" is aan te leren door te wijzen op de gevaren die opdoemen tijdens de refactoring. Onze methode die we hanteren in het verschaffen van inzicht is door al te wijzen op de potentiële gevaren vooraf aan de refactoring.

BEGELEIDING OP BASIS VAN ADVIES Specifiek in dit onderzoek richten we ons daarom met name op het educatieve aspect van refactoren, wat we benoemen als 'Procedural Refactoring Guidance'. Studenten worden bijgestaan door gedegen advies op basis van de gekozen refactoring en de staat van de code. Het belangrijkste doel is leren door te doen en de volgende keer fouten proberen te voorkomen.

ONDERSTEUNING DOOR INZET VAN TOOLING Voor het kunnen faciliteren van begeleiding is het zaak de nodige tooling daarvoor te ontwikkelen. Onze belangrijkste bedrage, beschreven in dit onderzoeksverslag is het uitwerken van een Framework voor het ondersteunen van risico gebaseerde refactoring begeleiding. Dit Framework biedt een uitwerking van de hierna volgende onderwerpen.

CONCEPT VAN DETECTOR CHAINING Om de toestand van de code te onderzoeken, hanteren we het principe van het uitvoeren van code diagnostisering. De diagnosestelling wordt gedaan door een keten van detectoren die elks op bepaalde voor aspecten van de code gaat letten. De interpretatie van de gevonden constateringen valt niet onder de verantwoordelijkheid van de detector zelf.

VERDICT IDEE Voor het onderkennen van potentieel gevaar introduceren wij de nieuwe benadering van redeneren over de toestand van de actuele code in relatie tot de te nemen voorgeschreven refactoring stappen. De uitkomst van dit redenatie proces is een uitspraak. Op basis van deze uitspraak formuleert de tooling een advies op maat aan de student.

FOWLER REFACTORING DECOMPOSITIE Elke refactoring die we (gaan) ondersteunen moet terug gebracht worden tot elementaire Microstappen van code wijzigingen. Deze Microstappen vormen de basis voor risico bepaling en daarvan afhankelijk de advisering. Omdat Fowler op een hoog granulatie niveau zijn refactoring procedures beschrijft, is het procedé bedacht hoe deze fijnmazige Microstappen te herleiden.

SCENARIO GEBASEERD ADVIES De uit te voeren Microstappen hebben effect op de toestand van de code, doordat ze verandering aan de code aanbrengen. Er is een mechanisme nodig om de scenarios te ontwikkelen die de mogelijke wijzingen en het gevaar dat ze daarbij opleveren in kaart kan brengen.

WORKING EXAMPLES Het beproeven van ons Framework wordt aangetoond door een prototype tool met working examples.

1

Introduction

Implementing good quality software is becoming yet more critical, as seen worldwide, the amount of code[1] expands rapidly. The need for more IT-based solutions, enhancements and repairs accelerates the growth rate even further. Refactoring software code is one of the powerful instruments to guard the quality of software. Researching the field of refactoring is not only scientifically interesting for developing sophisticated refactoring tools and techniques, but the appliance itself makes people less vulnerable to coding flaws to come.

1.1. Refactoring Foundations

Within this introduction, we present a small recap about refactoring for the less initiated reader. You can safely skip to the next chapters Fowler Refactoring basics (section 1.3) or Research Method (chapter 2).

1.1.1. Definitions

The origins of software refactoring and the term are unclear. According[2] to Erich Gamma [Fowler, 2018], the term 'refactoring' was conceived in Smalltalk circles, probably coined by Bill Opdyke and Ralph Johnson around the late '80s. However, in

[1] Software Volcano. Lecture slides from the University of Amsterdam. Authors Paul Klint Tijs van der Storm and Jurgen Vinju

[2] Foreword from Fowler's book about refactoring [Fowler, 2018]

the article from William Griswold and William Opdyke [Griswold and Opdyke, 2015], "Software refactoring was independently invented in the late'80s by two students in two research groups: Ralph Johnson's group at the University of Illinois and David Notkin's group at the University of Washington."

Actually, what does software refactoring mean? Refactoring is the process of changing a software system that does not alter the code's external behavior yet improves its internal structure. This definition is the compact version of two further definitions Fowler [Fowler, 2018] mentions in his book, depending on refactoring as a noun or refactoring as a verb:

Refactoring (noun) *is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*

Refactoring (verb) *to restructure software by applying a series of refactorings without changing its observable behavior*

A Fowler refactoring is characterized by -as he explained himself- "A disciplined way to clean up code that minimizes the chances of introducing bugs." Furthermore, "Each refactoring describes the motivation and Mechanics of a proven code transformation."

1.1.2. Motivation

In essence, when you refactor, you are improving the design of the code after have been written. Because of a better design, the software tends to be easier to understand and has positive effect on maintainability. Refactoring helps develop software more quickly and during the obligatory testing phases during refactoring, and as a result of refactoring, it may help to find bugs earlier. The software becomes less prone to errors. The cause for the programmer to execute a specific refactoring is what Patrick de Beer [de Beer, 2019] coins as the 'Refactoring Subject' and classified as what Fowler calls 'a code smell'.

Why refactor software:

- From the perspective of adding new functionality: *Refactoring is about preparing code for introducing new functionality*
- In you do not change code but only add new functionality (Open-Closed Principle), *refactoring is about restoring code structure*
- *Refactoring is about reducing complexity*. It may not always be evident to the inexperienced user that a particular refactoring lowers the complexity. For example, refactoring towards a Design Pattern like the Composite Design pattern, tends to increase the perceived complexity of the code for users that do not understand Design patterns or do not know how to refactor towards patterns. In his excellent book, Joshua Kerievsky [Kerievsky, 2005] tries to bridge this knowledge gap.

Novice developers occasionally try to refactor and build new functionality at the same time. Martin Fowler emphasizes this phenomenon by the two hats metaphor[3] lend from Kent Beck [Beck, 1999], as Fowler portrays, this kind of risky development is done by a developer wearing two different hats.

For those who need a Refactoring fresh-up, we have added a dedicated chapter Exploring Refactoring (Appendix D) to the Appendix.

1.1.3. Which code?

In order to determine the '*Refactoring Subject*', look for signs that suggest the need for refactoring. Those signs are bad smells in code. Bad code smells make code less maintainable and understandable. The designation of a "Bad Smell" was quoted in conversation between Martin Fowler and Kent Beck when to start and to stop a refactoring. Fowler's mother expresses this as follows: "when it stinks, change it".

In next source code fragment, the method *m(int x)* is the Refactoring Subject for the Rename Method refactoring.

```
1  public class B extends A {
2
3      public void m(int x) { // <--- refactoring subject
4        System.out.println("method m-int in class B");
5      };
6
7      public void m(Number n) {
8        System.out.println("method m-number in class B");
9      };
10 }
```

Listing 1.1: Refactoring subject example listing

The Refactoring Subject is part of a bigger picture, called Code Context. We concur with de Beer's definition for the Code Context. The Code Context can span all the code of the project; in this case, the scope is equal to the complete content of the AST.

1.1.4. When to?

Refactoring best practices:

- In general, plan for frequently refactoring in order to improve or recover the overall quality of the code.
- It is advised to plan code refactoring in preparation for new features. The suggested order is to prepare the application for new functionality by refactoring first. Once functional tests indicate that code behaves as it supposes to, only then is it safe to start developing new functionality. Functional tests are an integral part of good refactoring practice. Testing is a crucial ingredient within the Fowler Mechanics steps level (subsection 1.3.2) recipes.

[3] One hat stands for leaving existing code intact whilst adding new functionality (along with possible new test cases). The other hat stands for the restructuring of the code during refactoring activities instead.

1.2. Refactoring Examples

In this chapter, an introductory refactoring named "Replace Magic Literal" will be showcased. It is a relatively limited case refactoring, in the sense that it is a single dedicated task, that of the replacement of a numeric or string literal by a more descriptive symbolic constant. Currently, it is only listed on the website accompanying the second edition of Fowler's "Improving the Design of Existing Code" standard work.

1.2.1. Simple refactoring

The "Replace Magic Literal" refactoring is isolated from other cataloged refactorings. In isolation[4] means no other refactoring will be included in this refactoring, and other refactorings do not depend on this particular refactoring. We choose this type of isolated atomic refactoring on purpose for the benefit of simplicity.

The "Replace Magic Literal" exhibits the intent of the programmer to give special meaning to a literal. The purpose is to make the code more readable (regarding code maintainability to understand its purpose better).

Consider the following source code expression to calculate the circumference for a circle:

```
2 * 3.14 * radius
```

Obviously 3.14 has special meaning as a literal, representing 'π' in this case. We can resolve replacing the literal with a constant value or we can replace the literal by a function returning the value.

```
2 * PI * radius
```

The convention here is to formulate a constant value in capitals, as above PI, or alternatively, by replace it by a function like below pi()

```
2 * pi() * radius
```

The following potential naming hazard issues should be regarded for risk-based refactoring:

- For refactoring, appropriate naming is vital for its functionality; the name-giving of constants should agree to the intent of usage. The user should always be aware of the semantics of the literal to be correctly replaced; namely, the number 'pi' can also represent a TEX version.
- Naming conventions should also be honored. For example, if you encounter the FALSE constant in code, we may presume its value equals 0; a value unequal 0 we consider TRUE. Nevertheless to say that 1 or -1 is the magic value for TRUE, is a matter of convention.

[4] A refactoring that depends on other refactorings is a 'composite' refactorings. Those not depending on others are 'atomic' refactorings and work in isolation.

- Another example of intent is for example the constant MAXINT. Depending on programming language, system environment and operating system, MAXINT is not a fixed value. This is a possible hazard to inform the refactoring practitioner about.

- Do not overuse the replacement towards constant values. In the above example it seems rather unnecessary to replace the value 2 by a TWO constant. This could be a hint to be given to the student at the start of the refactoring

1.2.2. Intermediate refactoring

Not only naming of magic literals (as seen in previous example) is quintessential. In general, the naming of code elements like variables or methods is of importance. Whenever we pick explainable names, the maintainability of the source code improves. Maintainability is undoubtedly an excellent reason for renaming names. Renaming a method seems deceivingly simple but can be a potential source of trouble towards code integrity.
For instance, in the upcoming code example excerpt Nested Classes listing (Listing 1.2), one might be tempted to use the search and replace facility of your favorite source-code editor. Doing a global search and replace action for this example does not work because of other non-related *test()* method. It only works for the target method itself. The strategy here to follow is to replace the method callers besides the target method itself.

```
1  package test_inner2;
2
3  class Outer {
4     public void test() {
5        System.out.print("Outer class test()\n");
6     }
7
8    public void tester() {
9      Inner myInner = new Inner();
10     myInner.test3();
11     }
12
13
14
15   public class Inner{
16     public void test() {  //to be renamed
17       System.out.print("Inner class test()\n");
18     }
19     public void test3() {
20       test();
21          /* initially: resolved to test_inner2.Outer.Inner.test()
22           * renaming method, 'magically' resolves
23           * to test_inner2.Outer.test()
24           * however this leads to behavior preservation break
25           */
26       this.test(); //rename to prevent the compile/IDE edit warning
27     }
28   }
29 }
```

```
30
31 public class runner{
32   public void start() {
33   System.out.print("Renaming Inner.test() \n");
34   Outer myOuter=new Outer();
35
36   myOuter.test(); // call #1
37   /*
38    * Outer class test()
39    * Inner class test()
40    * Inner class test()
41    *
42    * after only renaming Inner.test() to Inner.test2() but fixing this.
       test()
43    * Outer class test()
44    * Outer class test()
45    * Inner class test()
46    *
47    * fixing the non referring test() to test2() within Inner.test3()
       resolved the issue?
48    * Outer class test()
49    * Inner class test()
50    * Inner class test()
51    *
52   */
53
54   myOuter.tester(); // call #2
55   }
56 }
```

Listing 1.2: Nested Classes listing

Renaming hazards

If you are developing within an IDE, code will be examined automatically for you after some modification. Tooling like Eclipse has automated build generation and internal AST processing at its disposal. This makes it possible for the IDE to give smart hints during edit time. Looking at the nested classes example again and suppose we do a rename refactoring, after the rename attempt from the inner class method *test()* to *test2()*, we would catch a compile error that says: "*this.test()* is pointing towards an undefined method". The error is plausible because *this.test()* referred to the original method and should be fixed to point to a new method name, i.e. *this.test()* should become *this.test2()*.

Now we have encountered an odd situation. Namely, as soon as we alter *Inner.test()* into *Inner.test2(),* magically as it seems, the inner class *test()* call refers to a new location after compile. Running this example confirms that the pointer shifted to the outer class *test()* method. A behavior preservation break without any warning has happened. Note that inner class *test()* is not a caller anymore to the renamed method. The only way out of this quest is to fix the inner class *test()* as well from *test()* into *test2()*.
During a joint working group refactoring session, we figured out[5] that both scope and

[5]Credits to my fellow graduates, refactor buddies Evert Verduin and William Wernsen

method caller investigation are applicable instruments to determine all references to the old method. It would be nice if tooling would suggest to us this possible hazard before even performing the refactoring.

1.2.3. Complex refactoring

There are several possible hazards thinkable for each possible refactoring. Let us, for example, take the case of the Rename Method refactoring again. We will look at the case of renaming an overridden method in combination with overloaded methods.

We speak of method overloading when two or more methods at the same class level have the same name but with different parameters. At run time, the compiler is capable of determining which particular method to invoke when called. When a class inherits from another class, we speak of overriding between methods with the same name, type and number of parameters.

The consequences after a renaming action are very diverse and depend on the given context. In our example, we are in particular interested in the case of a method renaming, constituting an abstract class hierarchy.

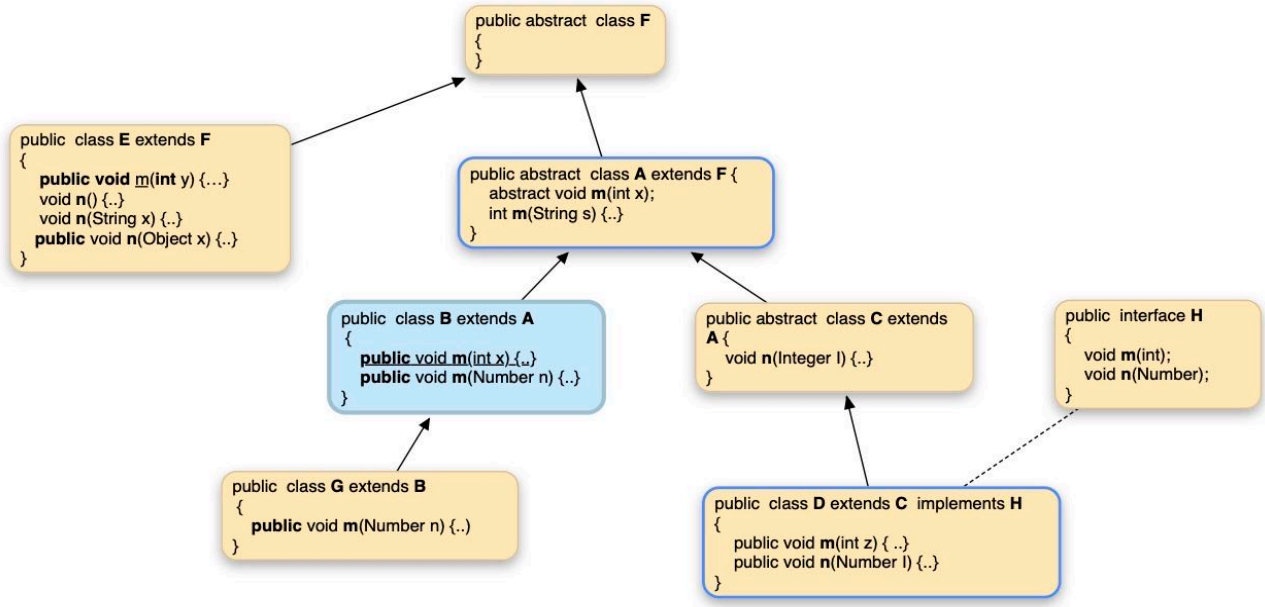


Figure 1.1: Abstract Class example

The intended Rename Method refactoring here Figure 1.1, is renaming *void m(int)* in class B into *void n(int)*. In this case, a cascade of rename actions should be executed to fix all kinds of compiler warnings. It would be nice to spot potential dangers in advance, even before the refactoring started. Renaming from method name 'm' to 'n' would accidentally break both overloading and overriding in class B, affects the

override in class D, and then again forces us to cope with the abstract method in class A.
The problem with abstract class methods is that the whole implementation hierarchy should be examined and handled. Fixing the abstract class case will only be part of the solution. In the case of renaming a method inherited from an interface, problems also arise to be addressed. Restore the overloading in class B, with the void m(Number) method also requires dealing with the same method in class G.

The migration Mechanics variant of the Change Function Declaration refactoring comes to the rescue. This figure Figure 1.1, also shows that class D, regarding method 'm' implements from interface H. For simplicity, renaming methods in interface classes is another scenario in which we also report interface H as an affected class. All interface class-related problems are out of scope in this example.

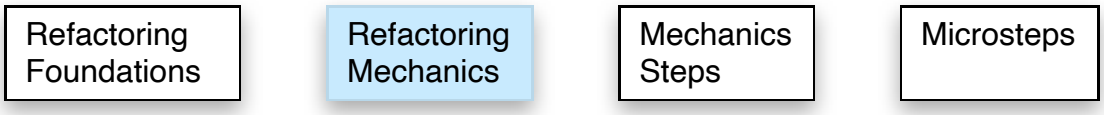
Regarding the migration Mechanics variant, the Change Function Declaration refactoring depends on other Fowler refactorings as a composite refactoring. When migrating, the Extract Function refactoring should copy the whole source method body code, and place them into a new method (albeit by a temporary name).
Important to know is that copying the whole body at once rules out the problem of variables declared inside the body getting out of scope but are in use within. Next, the copied body code then will be replaced with a call to the new method. With the Extract Function refactoring, effectively, we created a method-call indirection, but without renouncing the original method. We can test for possible behavior breaks because we have both the old and new methods in place.
Once we are confident about expected outcome, we can reverse the redirection by letting the method call to the old source method point to the newly renamed method.

When awkward at the step of actual renaming, the newly renamed method may clash with reserved Java keywords, with reserved method names like *main()* or redefine import naming definitions (like the PI in *Math.PI* from the aforementioned simple example). Be careful that the new name does not introduce unintended overloading or overriding. To get the point, there are a plethora of potential dangers lurking. This is why we need good concepts (and to implement them in tooling) to guide the student in refactoring.

1.3. Fowler Refactoring basics

To underpin our contribution, we would like to go through some basic principles in this chapter. Refactoring Mechanics is the foundation we build upon.

1.3.1. Mechanics level



Definition

For each refactoring, Fowler describes the motivation and Mechanics of a proven code transformation. According to him, Mechanics are a concise, step-by-step description of how to carry out the refactoring. In his book, we can read that he selected the Mechanics in such a way that they work most of the time, nevertheless the Mechanics may be varied as suited to own insight. In essence the Mechanics contain functional details about a specific refactoring. They originated from Fowlers' hand-written short notes on how he did the refactoring, but without explaining why.

Properties

There are specific properties that the Fowler refactoring Mechanics should adhere to:

- Safety first is key. The Mechanics for a refactoring should comprise small and safe steps. As Fowler states: "I've written the Mechanics in such a way that each step of each refactoring is as small as possible; take very small steps and test after every one". Fowler's approach will help people change code one small step at a time, thus reducing the risks of evolving your design.
- Mechanics should be easy to use. As a consequence, all the Fowler selected Mechanics in his book work pretty well most of the time (and is nice entrance for our guidance tool to trial on)
- Content should be comprehensive without too much technical level of detail. This helps improve refactoring skills and benefits to a good design of code.

Variants

Most of the Fowler refactorings contain only one variant of Mechanics, describing all the steps per refactoring case. Sometimes there are different strategies to follow for the same kind of refactoring. For example, the "Change Function Declaration" Refactoring elaborates on two variants: Simple Mechanics (which resembles a search and replace) is often suitable enough for standard refactorings. The more enhanced Migration Mechanics variant is a gradual refactoring implementation strategy for complicated cases, or if you run into trouble applying the simple Mechanics variant.

1.3.2. Mechanics steps level

Definition

Refactoring Mechanics consist of individual smaller steps; called Mechanics steps. According to Fowler, "*Refactoring is all about small behavior-preserving steps*". But what are steps in this regard? For our definition of a step we can refer to how Mechanics are represented. They are organized (and written down as such) as a bullet-wise list of text. Hence, all text belonging to a single bullet represents a Mechanics step.

The smaller these Mechanics steps, the better code changes are manageable (less risk and less error-prone). Manageable code allows to find bugs easier (by testing earlier and more often). One vital aspect of the Mechanics is that a Refactoring functionality remains unchanged until the level of the Mechanics steps. Within a single step we alter code structure but code will still be testable between each step.

Another benefit mentioned by Fowler is that introducing small steps (and therefore small code changes) "enables a tight feedback loop". In his opinion, "A feedback loop is key to avoiding lengthy debugging sessions". We not only concur with this statement but want to elaborate on this in that proper feedback or suggestions before the actual execution of a step may even save someone from debugging sessions at all.

Composition of Directives

Mechanics steps structure

Adopted from the "Move Statements into Function" Refactoring, we show following code excerpt below, to demonstrate that one bullet point line of text from Fowler: "*If the target function is only called by the source function, just cut the code from the source, paste it into the target, test, and ignore the rest of these Mechanics.*", can, in fact, represent more Mechanics steps at the same time. This single line of text composed of many Mechanics steps, must be split up into meaningful single Mechanics steps.

So the single bullet line of text reveals in fact more than one Mechanics step; the directives: perform/cut/paste are actually three Mechanics steps.

```
IF context condition ... THEN
   perform actions: CUT targeted-code-block
```

```
AND PASTE targeted-code-location
   STOP refactoring
END
```

Mechanics step directives

A more in-depth examination of the Mechanics reveals that the text resembles a kind of pseudo-language constructs to distinguish certain language elements we label 'directives'. We can distinguish the following types of directives from the example-excerpt: actions, control flow, context conditions, and instructions. For the sake of simplicity, we will discuss them very briefly for a better understanding.

With augmented explanation below we can dissect the above example:

Table 1.1: Example Mechanics steps

Directive	Example
Control Flow	The IF/THEN statement is control flow to steer which of the directives comes next
Action	Cutting and pasting of the code are actions that should be performed by the refactoring practitioner
Context condition	In case the target function is only called by the source function, fulfilling the condition, part of the Control flow directive.
Instruction	The Test directive and the Stop directive are instructions to steer the process of refactoring. Hints given also belong to this category.

Directive details

Control Flow

The fragment above contains conditional control flow because of the IF keyword. The conditions to be met can be anything, ranging from a situation in code to the outcome of earlier Mechanics steps. We see textual keywords like 'If', 'In case', and similar words typically usher control flow.

It is imperative to know that control flow is a mechanism to determine the execution order of directives. In complex refactoring situations, the directives sometimes call in the help of other refactorings to break down the complexity. We might bear witness of a single Mechanics step iterating over directives. For example, to find multiple occurrences. Words like 'Repeat', 'Switch', 'For .. do ..' tend to affect control flow.

Actions

The text that follows after the matching condition part of the IF control flow statement describes actual actions to be taken on the source code. Think of actions as directives that lead to transformations of the source code. In the following sections we will elaborate more on actions because transformations are the cause of breaking behavior,

but not every code transformation will lead to breaking it. We want to investigate the relationship between transformations (and the effects it resorts) further in this study.

Context Conditions

In this regard, the actions mentioned here are only 'allowed' under the context of the condition, as we have seen in the example, the condition to be matched follows after the IF keyword. Sometimes Fowler is very explicit to look for specific existence or states in code. Words like 'Check' or 'Find' urge the user to examine the context of the code before proceeding with the refactoring.

Instructions

They serve the purpose of giving feedback, reconsidering the refactoring, or request the user to do a compilation or functional test of the source code. Anything not directly related to the above directives may be regarded as instruction as well.

2

Research Method

In this chapter, we are introducing our main research question and supporting sub-questions.

2.1. Research questions

2.1.1. Main research goal

The purpose of this research is: to what extent can Fowlers refactoring Mechanics be comprehended and molded into a suitable solution to enable a risk-based 'how to refactor stepwise' guidance. This goal is relevant because tooling that delivers advice like a teacher would do can make that difference in the proper execution of a refactoring. Risk-based notifications make sense from an educational perspective. Generating warnings or point to potential dangers make the students more aware that refactoring remains a risky business if not performed with care.

Main research question *How can we conceptualize and deploy a system that delivers contextual-based refactoring guidance pertaining to a small selected set of Fowler's refactorings, by the notion of code state diagnostics and extendible to support progress monitoring?*

Subquestions RQ1-RQ4 have been defined to support the main question

- Fowler Refactorings decomposition (subsection 2.1.2) (RQ1)
- Devising detectors (subsection 2.1.3) (RQ2)
- Risk-based advice (subsection 2.1.4) (RQ3)
- Monitoring integration (Future Work) (subsection 2.1.5) (RQ4, optional)

Topics of interest:

- (Fowler) Refactorings:

  Introduction (chapter 1), Fowler Refactoring basics (section 1.3), Analyzing refactoring Risks (chapter 4)

- Risk-based refactoring Guidance:

  Risk-based Refactoring (chapter 3), Risk-based Guidance Use Case (subsection 2.3.2), Risk-based Refactoring Process (subsection 2.3.1), Guidance staging Model (subsection 6.1.2)

- Code diagnostics and evaluation: Source code Diagnostics (chapter 5)

- Tooling facilitation:

  Prototyping the Framework (chapter 6), Assisted tooling (section 3.4)

2.1.2. Fowler Refactorings decomposition

The refactoring Mechanics Fowler describes have been written in a natural language; albeit easy to read, but quite difficult to master. A simple excerpt like "check for any references" troublesome the student's intention to do the refactoring, because what does the term "references" imply, and foremost what needs to be done to fulfill this designated statement?

What we want to achieve, for at least one refactoring, is to break down the refactoring Mechanics, per selected refactoring, into smaller manageable tasks. This raises an interesting question, concerning detectors; do we notice repetitive patterns arising when we dig into a list of refactorings? It would be nice to see the pattern of repetitive usage of some detectors or the same set of detectors. Do we discover detector chaining calls that are common between refactoring Mechanics in future work?

RQ1 *How can we define a framework for decomposing the Fowler refactoring Mechanics into composable actions to enable guided instructions on proceeding based on the actual code context?*

Topics of interest:

- Decomposition of Fowler refactorings:

  Mechanics level (subsection 1.3.1), Mechanics steps level (subsection 1.3.2), Microstep level (section 4.1), Matrix of Microsteps (section 4.2), Matrix properties (section 4.3)

- Detectors and chaining of detectors: Detectors (section 5.1)

- Guidance and actual code context:

  What-If concept (subsection 5.2.1), What-If develop recipe (subsection 5.2.2),

2.1.3. Devising detectors

Detectors are essential to the process of determining the current context of the code. When the student tries a particular refactoring, the system should examine the context of the code in order to generate tailored and context-based advice. The set of detectors combined should cover a broad scope of functionality required to get a good overview of the code construction the system has to deal with. In this respect, we talk

about a generic (refactoring independent) set of code context detectors because each detector will be given a specific job, fit for detecting a specific code construct property.

A small inventory of detectors for one or two refactorings will be suitable enough as proof of concept for this study. It is desired that we can configure detectors in such a way so that we can reuse or share (a group of) detectors between refactorings. Detectors that are extensible (using object-oriented capabilities of the build language) will be assumed to promote reusing and sharing. Diagnostics about the state of the code under investigation should ideally be allowed on-demand at any given time during the refactoring process. In the Monitoring refactoring progress, step (6), as depicted in Figure 2.1, tooling should be capable of generating source code diagnostics on purpose.

RQ2 *How can we devise a generic set of code context detectors for the selected refactorings that will enable the tooling to assist in guiding the student?*

Topics of interest:

- Detectors: What-Ifs and Detectors (subsubsection 5.2.1), Detectors (section 5.1)
- Guiding assistance: What-Ifs (section 5.2)

2.1.4. Risk-based advice

The concept of risk-based advice is based on two mechanisms; first, we must run code diagnostics to gain insights about the state of code. Secondly, we draw conclusions regarding to these risks. This means we need to be aware of the dangers and how to treat them accordingly. Deciding how to react depends on the encountered refactoring scenario and the actual code context.

RQ3 *How can we devise assisting the student with advice by reasoning about the conditions to be met for accurate refactoring diagnostics?*

Topic of interest:

- Reasoning about advice:

  What-Ifs and Detectors (subsubsection 5.2.1), What-Ifs (section 5.2), Verdict Idea (section 5.3)

2.1.5. Monitoring integration (Future Work)

Suppose we want to guide the students during the whole execution of the refactoring. In that case, we need to deliver feedforward advice to the student, that is, guidance even before any actual code alteration and feedback generated between each Mechanics step. If we look at the refactoring workflow, as depicted in Figure 2.1, this question (IV) resembles steps (6) and (7). Therefore, it is recommended to continually monitor the student's efforts and track the progress between all the refactoring steps.

As a benefit, by tracking the efforts, we can produce more specific feedback and ignore ill-considered advice. Consider for example the existence of two references pointing to a variable that needs to be extracted as part of the "Extract Field" refactoring. In any case, all references should be dealt with. The system should be able to measure the progress and when done fixing the references, it can tell the student to carry out the next refactoring step.

RQ4 *How can we integrate monitoring functionality into the toolset to assess the execution of the manual refactoring steps?*

Subquestion RQ4 is considered optional, to be regarded as Future Work!

2.2. Research scope

In the context of this research, only the Change Function Declaration (simple Mechanics variant aka Rename Method) refactoring and Extract Function refactoring are initial candidate refactorings (prioritized in that order and parts of Extract Function optionally).
The Change Function Declaration - Migration Mechanics variant is a composite refactoring containing the Extract Function Refactoring and Inline Function Refactoring.

The language to be supported is Java only. The aim is to support at least the fundamental language constructs of Java until version 8

Complex language constructs such as: dynamic binding, reflection, Steaming, lambda expressions and Generics, for example, are considered out of scope.

2.3. Refactoring Guidance research result

Our contribution is presented by a set of concepts we call '*Risk-based Refactoring Guidance Framework*', which is divided into the following chapters:

- Risk-based Refactoring (chapter 3); introduction of the terminology and the causes of risk.
- Analyzing refactoring Risks (chapter 4); extending Fowler Refactoring concept to support the identification of risks.
- Source code Diagnostics (chapter 5); the core of the Framework to make it achievable to reason about refactoring dangers.
- Prototyping the Framework (chapter 6); for conceptualizing the architecture of the Framework and presenting a working solution.

Risk-based Guidance Refactoring is embedded within a Risk-based Refactoring Process (subsection 2.3.1), introduced here to determine the scope of this study. We provide a use case in Risk-based Guidance Use Case (subsection 2.3.2) to accompany this study by a working example throughout this thesis.

2.3.1. Risk-based Refactoring Process

This thesis addresses the appliance of code refactoring in the field of 'Procedural Guidance research'. This research deals with the educational aspect to make students aware of how complex code refactoring is and, foremost, how to learn from it by giving guidelines. To accomplish this task, we need tooling to deliver a well-grounded platform that accompanies students in carefully executing the refactoring.

Tom Mens [Mens and Tourwé, 2004] laid out the foundations of a procedure to refactor in a controlled manner. Patrick de Beer [de Beer, 2019] continued on the process of refactoring by augmenting and encapsulating the possibility for cyclic feedforward (before the refactoring) and feedback (during progress).

We want to further conceptualize and extend the mechanisms for producing advice based on risk. These risk-based diagnostics can be applied to the actual code context. Because refactoring is a complex matter, the contribution of this thesis is essential. With his RAG[1] implementations for two Fowler refactoring candidates, Patrick de Beer introduced the foundations for risk-based diagnostics. We further supplement and substantiate Risk-based Refactoring by developing a Framework. For example, the concept of the Microstep (later discussed at Microstep level (section 4.1)) is a new addition to the theory.

Extracted from Patrick de Beer's thesis, the whole process-cycle for a controlled refactoring as depicted within the Refactoring process (Figure 2.1), contains two smaller cycles between the dash-lined area, as a demarcation for our study. These cycles represent feedforward and feedback cycles. Feedforward is this context means; to advise on advance before any code transformations, whereas feedback happens during the refactoring actions. Our main focus is the Risk Notification (4) step.

Refactoring process explanation: As one can see in step (3), the student is responsible for selecting a supported refactoring to be applied. Steps (4 and 5) give feedforward advice to the student. The shortest cycle [3,4,5,6,3] denotes when the student performs a refactoring step but wants to try another refactoring. The more extended and interesting cycle [3,4,5,6,7,5,..] is when the student gets stuck or is heading in the wrong direction for a solution. Time for feedback!

The central part of this study is the feedforward advice at step (4) Risk notification, where detection for potential risks for a selected refactoring takes place. What is considered risk for refactoring will be elaborated on in more detail throughout this thesis.The difference between risk notification (4) and refactoring instructions (5) is that the latter purely addresses how to do the refactoring.

2.3.2. Risk-based Guidance Use Case

This research aims to lay down a conceptual basis for guiding students during refactoring, as endorsed by the Main research goal (subsection 2.1.1).

[1] RAG is an acronym for Refactoring Advise Graph

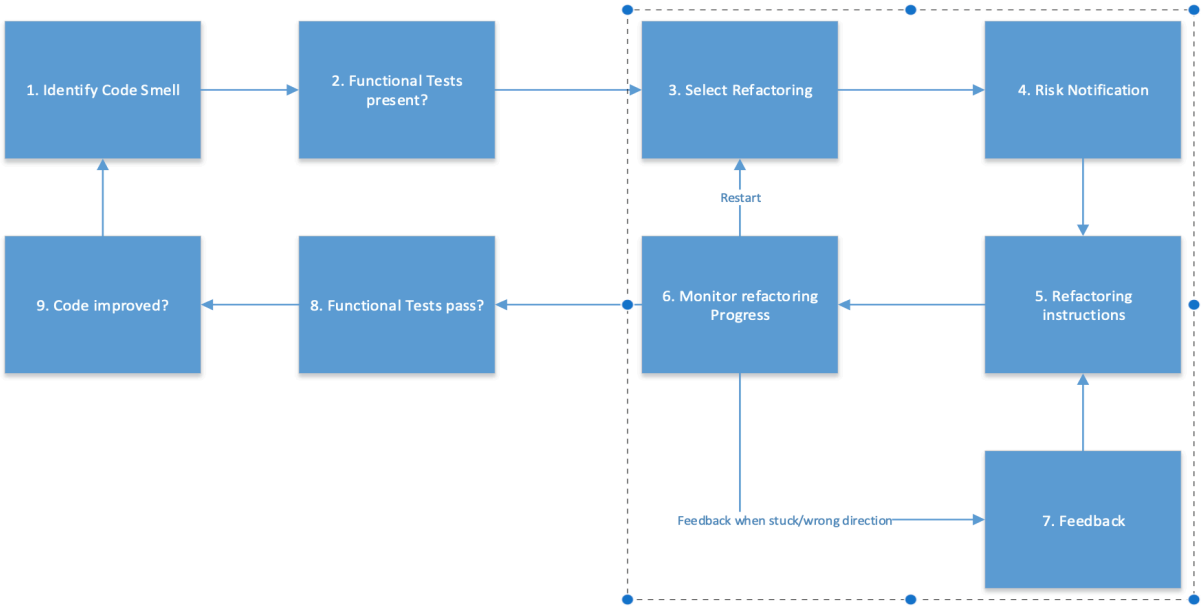


Figure 2.1: Refactoring Guidance Process

Let us regard our contribution with an example Use Case scenario; Renaming a method Complex refactoring (subsection 1.2.3) source context example. In this case, the student, again, wants to rename a method *m()* but this method has been declared abstract in a superclass.

- The user selects the method and asks the system for advice. Many issues can arise when renaming a method. One of the threats is imposed by our rename action. For the refactoring to succeed, all the issues should be avoided.
- Based on the given advice by the system, the user can react by checking all the affected abstract classes. The remedy for renaming methods declared abstract in a superclass, is to apply the Rename Method refactoring to all the affected methods.
- To yield all affected methods, those in real danger if we proceed the refactoring, the system needs to evaluate which candidate methods are in real danger.
- Before the evaluation, the system needs to find all the candidate methods possibly exposed by the danger; done by running a code diagnostics. Examining source code is a job for our detector(s).
- The task, to decide if a method is in danger, has been appointed to a dedicated What-If. It gets fed from output by detectors to verdict about the encountered situation.
- The What-If also has the dedicated task to provide the advice report to the student, as part of the guidance.
- Advice is textual information about detected dangers. Guidance is the process of supplying the generated list of advice to the student.

The following sketch Figure 2.2 shows a coherent overview[2] of the above-used terms. These terms have been addressed and explained in detail throughout the thesis.

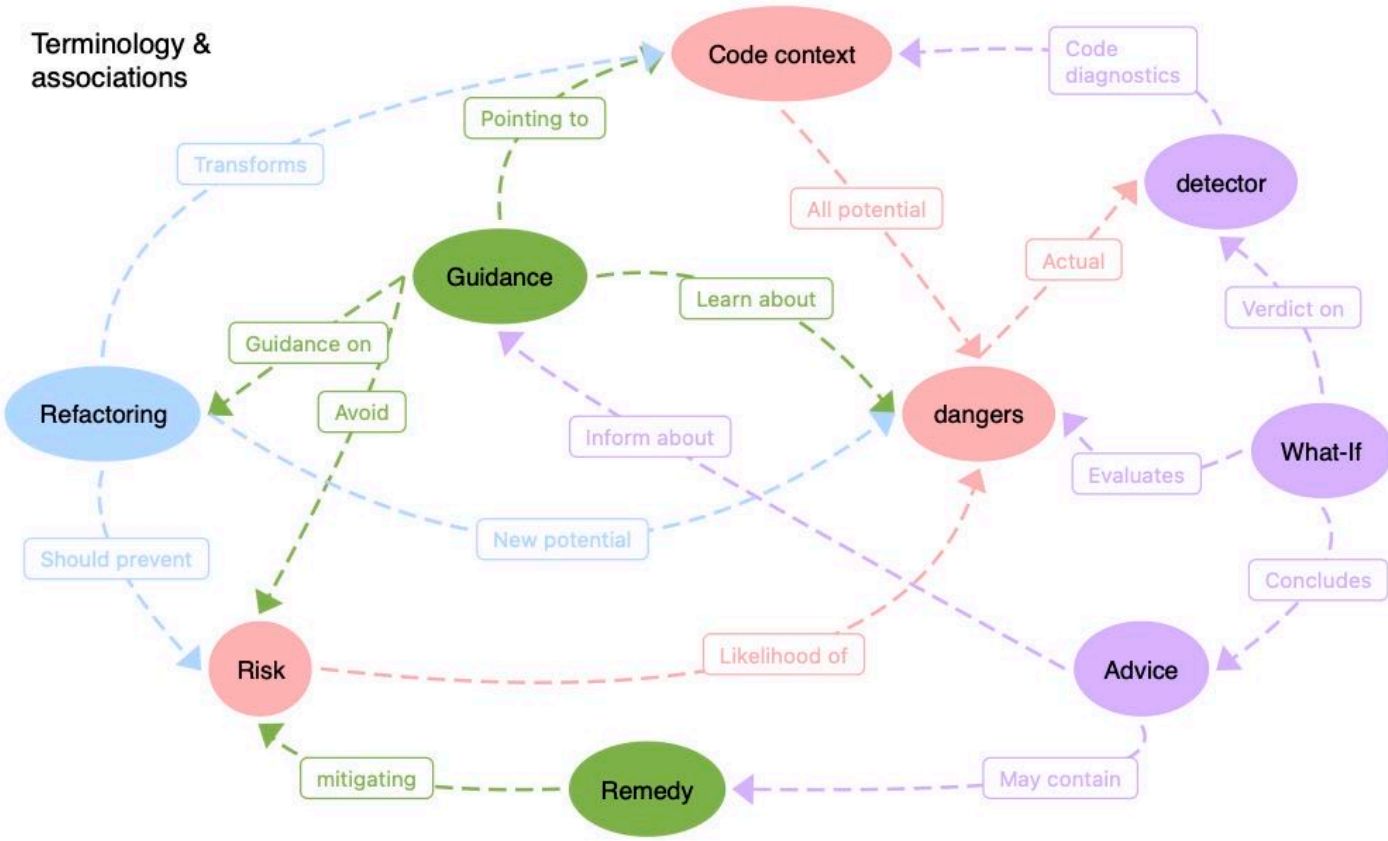


Figure 2.2: Risk-based refactoring Guidance

2.3.3. Reasoning about dangers

For risk-based refactoring guidance, we need to know the possible risks in and causing dangers advance. What Risk-based Refactoring is, what Risk is and what Dangers are, is part of further elaboration at chapters Risk-based Refactoring (chapter 3) and Analyzing refactoring Risks (chapter 4). The first chapter introduces the novel approach of taking risk-based refactoring into account and how to prevent risk every time during the guidance of the student. We will also briefly discuss risk causing factors and how to mitigate them.

The second chapter augments the Fowler Refactoring Mechanics, with identification of risks based at the Microstep level (section 4.1). These Microsteps can be aligned into a Matrix of Microsteps (section 4.2), based on scope and language constructs. The elegance of our solution, to determine risk, based on Microsteps, is that we abstract away from the many refactoring Mechanics. Concerning this study, we focus primarily on the 'Method column' related Microsteps because of scope. Of particular interest

[2]Colors are used for clarification and to group some terms with strong underlying relationship. The direction of the dashed lines can be interpreted as follows: a Refactoring transforms the Code context

is the method related Change Function Declaration calling the Inline/Extract Function Fowler refactorings.

Once we have identified the Microsteps for the refactoring and are aware of the possible dangers, we can evaluate the state of code, for which possible dangers are real. We have developed some concepts to facilitate the process of reasoning about dangers.

The following three major concepts are relevant to build the machinery of reasoning about dangers:

- *Detector chaining tree*[3]; a set-up in which detectors work together
- *Detector Layer model*[4]; the interaction patterns, describing the cooperation between all constitution types and subtype detectors
- *Detector Building Block*[5]; the mechanics[6] of a single detector

The above concepts are all detector-related. The figure Figure 2.3 expresses the relationship between them.

The detector chaining tree construct delivers the mechanism to be able to do Source code Diagnostics (chapter 5). The enabling element to identify real dangers in code is the concept of the What-Ifs (section 5.2). The What-If verdicts about the state of the code, as we will discuss at Verdict Idea (section 5.3). We elaborate on the Verdict process (subsection 5.3.1) and the concept of verdict levels.

The Source code Diagnostics (chapter 5) chapter is the most fundamental part of the study. The What-If concept (subsection 5.2.1), how to derive What-Ifs, is discussed in What-If develop recipe (subsection 5.2.2). We also talk about the rationale and concepts of Advice and Advice templating, as introduced at Risk-based Guidance Use Case (subsection 2.3.2).

Besides the Detector chaining tree, we individually discuss the type of the node elements; how to derive them, the internal logic of each detector type and the black box notation method to describe the individual tree elements sufficient to know what dangers are covered.

The chapter Prototyping the Framework (chapter 6), proves our concepts with two Solution (section 6.3) examples. We will also dive into the concept of our applied AST in Conceptualizing the AST (section 6.2), how to program the detectors. We accompany our Framework with a possible layer-based Envisioned Architecture (section 6.1), how to realize the Guidance Tooling.

Curious about an example Detector Chaining Tree? Jump to Detector chaining concept

[3]Image borrowed from figure Figure 5.5
[4]Image borrowed from figure Figure 5.6
[5]Image borrowed from figure Figure 5.3
[6]Not to be confused with the Fowler Mechanics.

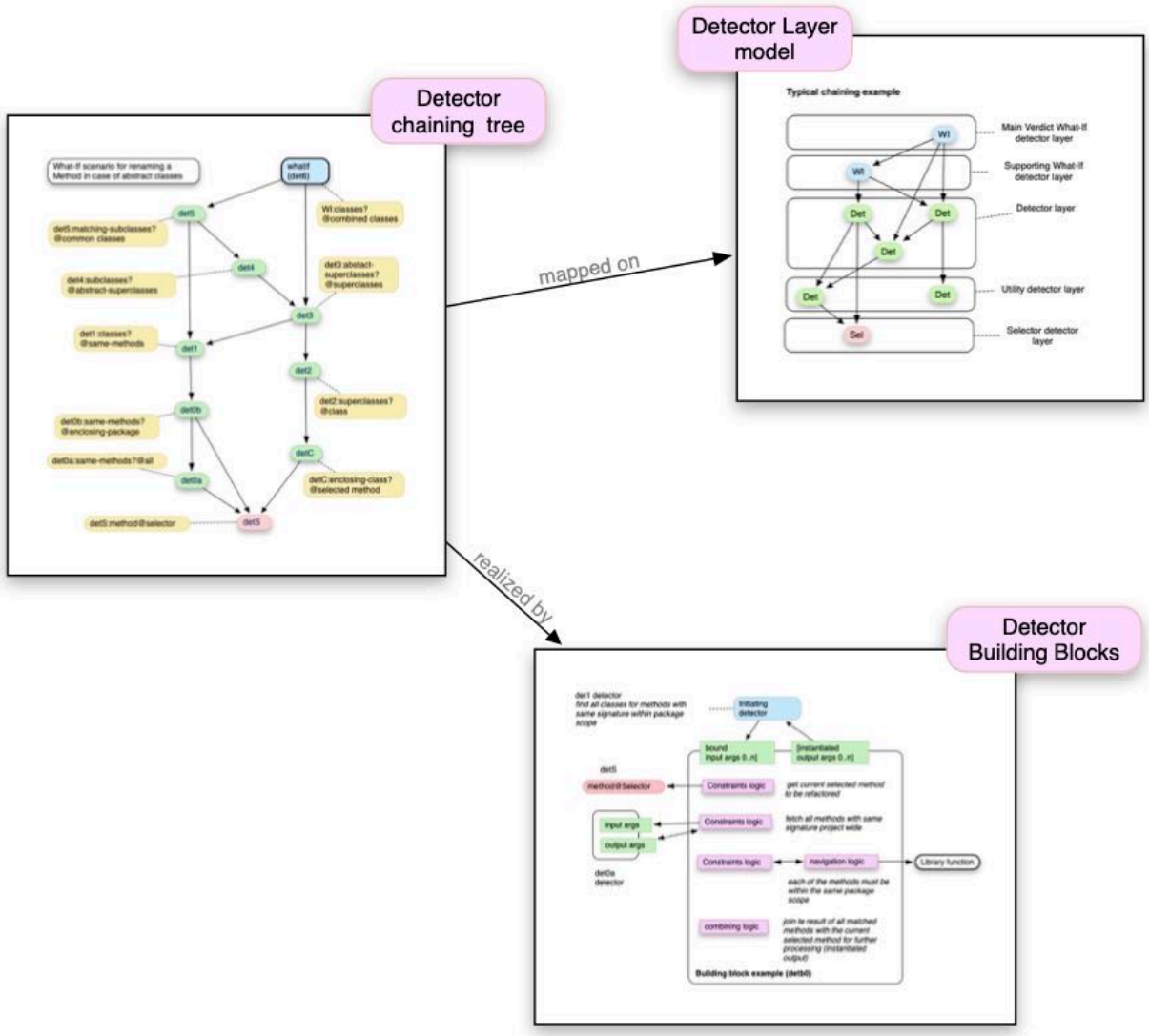


Figure 2.3: Detector chaining, layering, building-block concepts

(subsection 5.1.5) for the explainer and to EF-refactoring demo 2 (subsection 6.3.2) to see our assisting tool implementation in action.

3

Risk-based Refactoring

3.1. Terminology

Our goal is to implement risk-based refactoring, that is, taking the notion of risk into account during the refactoring process. Aim is to inform the refactoring practitioner about actual risks based on detected dangers. However, the terminology about risk and danger, potential and actual or possible risks, need to be explained briefly. The adjectives 'possible' and 'potential' are interchangeable, so possible risks are potential risks.

Risk *The chance or likelihood of actually being exposed to something or undergo the effects of something. That something can be a hazard (that causes harm) or likewise danger (danger is a synonym for hazard but with generally less inflicting damage).*

Possible risks

Exposure to risk can be caused by bad execution of the refactoring steps. Potential risks are a property of the refactoring. Changing code can lead to semantical or syntactical or functional disruption of the code. Program behavior breaks may or may not got to be detected by the compiler. Which means we have to introduce mechanisms to detect those dangers.

Actual risks

Analysis of the project code subjected to the refactoring determines if the actual risk is present. We want to point the student to the actual risks before inflicting harm, meaning that actual risks might happen for specific refactorings based on the code-transforming actions carried out. Actual risks are what we might also call real risks.

3.2. Risk factors

3.2.1. Technical error causes

For every refactoring, there is always the possible risk of the user making mistakes unintentionally. We may or may not be able to prevent these errors from happening. Then again, there is always the risk of getting exposed to non-compilable code. This category of error is at the syntactical level. However, these mistakes can be coped

with, with the help of the detailed warnings, from the compiler itself. The same applies to semantic errors, which adhere to the syntax of the language. For example, a common mistake is not to initialize variables properly. In most cases, the compiler catches these semantical errors as well.

3.2.2. Functional error causes

Breaking expected behavior is very dangerous to the program. Nevertheless, functional errors are difficult to catch without awareness of the dangers that caused them and how to treat them accordingly. Fowler suggests that proper testing should be the mechanism to validate the functionality of the code. If one of the functional tests fail, we can speak of the existence of behavior preservation breaking errors.

The risk type we want to tackle are those mistakes that break the functionality of the program; this is the category of behavior preservation breaking errors. The code itself is compilable and testable, but here the outcome is not what we expect, or the functionality is not what is expected even when all tests pass.

3.2.3. Fowler Mechanics related causes

Fowler refactoring Mechanics often lack the necessary depth for both inexperienced programmers and tool implementors. A majority of the consisting Mechanics steps merely describe what to refactor but leave out how to do so. Sometimes a Mechanics step is too rudimentary. For example, within the Inline Function refactoring, Fowler advises to stay away from the refactoring when polymorphism comes into play. But why shouldn't we attempt to go into more detail about what the dangers are and to explore the possibilities when or when not to refactor in different scenario settings?

The necessity to guide students to avoid possible risks when refactoring stems from the following encountered issues, those steps either are:

- **too coarse-grained**; steps, in general, are too high-level orientated and should be broken down in several sub-steps; in fact, this is the reason we opt for Microstep level (section 4.1) discussed in the next section; small and repeatable steps that are small enough to determine and relate to the potential risks when involved;

- *sometimes* **too abstract or too vague**; the following (Change Function Declaration) excerpt is an example about the difficulty for students to grasp without further details: "If you're changing a method on a class with polymorphism, you'll need to add indirection for each binding. If the method is polymorphic within a single class hierarchy, you only need the forwarding method on the superclass. If the polymorphism has no superclass link, then you'll need forwarding methods on each implementation class."

- **need expert matter knowledge**; refactoring is inherently complicated, refactoring requires refactoring skills and experience;

- **language-specific** *(regarding to description and result)*; Java does not support the concept of nested functions. The Extract Function refactoring solution is

different for Javascript than opposed to Java. To give proper advice, you need to be aware of the language under investigation.

Let us give an example of issues when dealing with a Fowler refactoring. As an illustration the Change Function Declaration refactoring, one of the Mechanics step state:

1. "*If necessary, refactor the body of the function to make it easy to do the following extraction step*"

   Nevertheless, how should we interpret this? In this case, Fowler indicates preparing the function's body by a precautionary step to group the affected variable declarations together near the extracted code. In fact, this step is opting in another refactoring, namely the Slide Statements refactoring.

2. "*If the extracted function needs additional parameters, use the simple Mechanics to add them*"

   When do we need to add parameters in the first place? And if so, we have a choice between exercising either the Split variable refactoring or, if confronted with too many parameters, we might consider the Replace Temp with Query refactoring.

3.2.4. Defective refactoring

The intricacies of a Rename Method refactoring on code forming the Decorator Design pattern, where refactoring is not that obvious, is given as an example by Jason McC. Smith [Smith, 2012]

As he states, an ill-considered change (on the method used in the Trusted Redirection[1] part of the Decorator) is quite harmful as it undermines the composition of the pattern ending up in a malignant Decorator design pattern. The danger concerning software quality here is the *architectural decay* caused by defective refactoring.

3.2.5. Other quality-related error sources

Out of scope for this thesis, but other risk candidates concerning code quality affecting maintainability are: code churn, merge conflicts, bugs, build breaks, cost consuming, and time-consuming aspects of the development process because of mandatory refactoring.

3.3. Risk mitigation

Risk represents the likelihood of encountering hazards. Thus, if we want to prevent risk as much as possible, we must be aware of the possible dangers at a refactoring. Although we know the refactoring steps to be taken beforehand, we do not know which potential dangers are becoming real unless we have a clear picture of the code state.

[1] With Trusted Redirection, we have two similar methods in dissimilar objects related through sub–typing. The appliance of Trusted Redirection is one of the EDP building blocks to compose the Decorator Pattern

To prevent actual risk, we need to guide the student with advice about possible dangers, and the guidance should include identifiable risks, best practice advice, and warnings about common pitfalls. Our mantra is: if we can avoid risks, we increase the quality of the refactoring outcome.

3.4. Assisted tooling

3.4.1. Support

The long-term goal we want to achieve is a tooling implementation to assist in the refactoring guidance process.

What benefits can we gain by deployment of tooling?

- Bridging the gap between complexity and the lack of knowledge
- Increasing the usability aspect by acting as a virtual tutor

We do not want to enable fully automated refactoring on behalf of the student. See Limitations (subsection 3.4.2) why not.

One of the software design principles is to strive for high cohesion and low coupling. When refactoring, the efficacy stretches out beyond local modifications of code. Tooling can help to sustain an overview of the systematics of refactoring. With the aid of tooling, we want to deliver a gentle introduction to the art of code refactoring. Students will be getting solid advice based on their choices and easy-to-follow steps involved with a particular refactoring, along with the impact or risk these actions impose. The main goal is learning by doing and trying to avoid mistakes next time.

For teaching purposes, usability is an instrumental quality aspect. When the tooling acts as a virtual tutor to the student, they benefit by:

- gaining insight into the process of refactoring;
- getting information about possible consequences that can occur when the student wants to perform the refactoring, like hazards (damage to functionality), pitfalls (or common mistakes), and other oddities ;
- retrieving feedback on actual refactoring work by the student;
- incorporated expert knowledge; learn to refactor from best practices;
- offering a playground for What-If questions, what happens if I do this or that;
- being informed about the consequences in terms of risk diagnostics.

Ideally, these features will be the target of future assignments. We plan to work on some of the aspects mentioned and the technical challenges to push usability forwards.

3.4.2. Limitations

When the code base is (too) oversized, or the refactoring becomes too complex (for example, nested Refactoring complexity), even refactoring with the aid of a tool can become quite cumbersome. Especially novice users may not understand the intricacies of complex refactoring and therefore are not willing to let the tool do the job. Current refactoring tooling seems to be more targeted towards a professional audience who already understand and can pinpoint the implications when refactoring [Bečička et al., 2007].

This way, we do not want to enable fully automated refactoring as part of the deliverables. Besides this, this kind of tooling has issues on its own: Issues with tooling in the Appendix.

4

Analyzing refactoring Risks

In previous chapter Fowler Refactoring basics (section 1.3), Fowler's principles of refactoring Mechanics came across. In this chapter, we want to substantiate why we need another layer of detail, called Microsteps, along with the acquaintance of the concept of applying What-if based diagnostics.

4.1. Microstep level

If we look at the picture Figure 4.1, we have now arrived at the level of the Mechanics step processing into Microsteps.

4.1.1. Definition

A single Mechanics step can be composed of even smaller steps, just like the composition of a molecule containing atoms. A Microstep represents our lowest level of decomposition to measure the impact of change. At this level, we analyze the code for potential refactoring risks by identifying potential dangers. Every Microstep may involve a series of transformations, but a single Microstep will be regarded as a (transactional) unit of work.

The way Fowler sets up Mechanics steps is that they honor the working state of code. After refactoring a Mechanics step, the program should still be compilable and testable because testing is the vehicle to guarantee code behavior preservation.

From the perspective of a Microstep however, by default, we change the structure of the code in such a way that we are allowed to and might temporally break behavior preservation. This implies that after execution of a Microstep testability is not guaranteed. Code quality validation should be suspended after completing the whole refactoring.

4.1.2. Why Microsteps?

At the level of a Fowler refactoring, we can speak of reusability when other Fowler refactorings are involved. The Change Function Declaration refactoring is an example of a composite refactoring calling in aid of another refactoring. However, at the level of the refactoring Mechanics, the individual refactoring Mechanics steps are very case-specific, and therefore hardly reusable.

Our proposed solution is to decompose the refactoring beyond the Mechanics steps, to the level of Microsteps, for the purpose reusability.

Microsteps are independent of the Mechanics, except that Microsteps manifest[1] within Mechanics. Microsteps do not need to know or even care about the refactoring configuration it is part of. Because of this detachment, Microsteps are suited for reuse. Since we only support a limited number of Microsteps, analyzing dangers should be less arduous than analyzing dangers for the Refactoring Mechanics steps, of which we have numerous more. We argue about the advantage Microsteps have over Mechanics steps in the subsequent sections why.

4.1.3. Mechanics versus Microsteps

As seen with the Change Function Declaration refactoring, one Fowler refactoring can have one or more Mechanics variants. Mechanics consists of at least one descriptive Mechanics step, and Mechanics steps can be shared between multiple Mechanics, hence the n-m relationship between Mechanics and its constituents steps. We noticed the 'reuse' of Mechanics steps., for example, the Mechanics Step "finding a call to a function" is often found in several Mechanics. Not only do we perceive reuse of Mechanics steps but also and foremost at the level of Microsteps. With only a limited set of different Microsteps, we already expect to serve a notable degree of Mechanics steps, analog to how molecules relate to atoms (with only a small set of atoms, we can arrange numerous molecules).

The following image Figure 4.1 visually shows the above-described occurrences.
Benefits of Microsteps sum up: Table 4.1

4.1.4. Source code transformation

[1]The analogy between the Mechanics and Microstep resembles the relationship between molecules and their limited constituent number of atomic elements.

Figure 4.1: Fowler Refactoring Microsteps hierarchy

Table 4.1: Microsteps benefits

Microsteps	Mechanics Steps
Limited set	Huge (infinitely possible) set
Cover all dangers	Fowler does not incorporate the notion of risk
Simple and well defined	Each step can be very complex and vague

Small-grained transformations

In an article about applying impact analysis in the refactoring process [Mongiovi et al., 2014], the term 'Small-grained transformation'[2] was brought up. Melina Mongiovi suggests decomposing a refactoring (being a set of coarse-grained transformations) for a program into small-grained transformations to analyze the impact of each one separately in the resulting program. According to Mongiovi, the reason for decomposition is that it makes the process of analysis simpler. To measure the level of impact, Mongiovi uses a limited set of small-grained transactions.

Our study proposes the term 'Microstep' instead of small-grained transformations, though, with the same aim, namely strive for safe refactoring, our approach angle is different. Mongiovi measures the impact with aspect-oriented byte-code comparisons between the program before and after transformation to determine the impacted subjects. We, on the other hand, want to achieve safe refactoring by running comprehensive code diagnostics.

Microstep AST operations

The role of a Microstep is to divide the refactoring into easily identifiable and manageable modifications to the AST of the code. The nodes in the AST form a tree representing the program, and a single node represents a language construct. We can add nodes to the AST, remove nodes from the AST or change the internal properties of an existing AST node. Because the number of language elements is limited, the number of Microsteps is also limited, depending on the source-code language.

We suggest supporting three basic operations and one convenient operation on nodes:

[2]Operations listed are: Add – or Remove a method, Change the modifier of a Method, or the body of a Method, operations like Adding or Removing fields and operations that Change the Field modifier or initializer.

- ADD operation, a fundamental operation to add an object (class, interface), attribute (field or var), or adding a function (method or constructor) to the AST tree
- DELETE (i.e. REMOVE) operation, as the reverse of an ADD, removing type, attribute or method-related nodes from the AST tree.
- CHANGE operation, facility to alter the internals[3] of affected AST, without the explicit need to replace the entire node(s) itself. Although a Change transformation can theoretically be seen as a Delete followed by an Add with new data, the Change operation supersedes the Delete and Add sequence. See for more explanation below at "Changing versus Replacing"
- MOVE operation, regarding this derived operation, a move can be presented as the sequence of a DELETE and ADD operation, within the same transaction boundaries.

Changing versus Replacing

In theory, a Change operation resembles the removal of the old node, to be replaced by a new node with new properties. But consider that deleting a node has implications when the node itself represents a tree of child nodes. Changing the current node is much more efficient than replacing whole tree parts.

Take, for example, the addition of a modifier to a method (making a method public) or removing the const keyword from a field. Instead of removing all child nodes to copy them later and attaching them to the parent node, we want to change the nodes responsible for the modifier language construct.
Another relevant argument not to substitute change by delete and add is when the operation takes place on a method node. When you remove a method, all references to the method node (not necessarily child nodes, like the case for method calls) become invalid.

4.2. Matrix of Microsteps

One can imagine that too many Microsteps would severely reduce the applicability to use Microsteps as building blocks for refactoring risk determination. Restricting the number of Microsteps helps to govern the balance between completeness and workability.

A limited set of Microstep matrix members, in relation to the language concepts in use, also offer the advantage to treat each Mechanics step as a reoccurring occasion of one of those Microsteps members from the set. All these Microsteps are member of The Matrix (subsection 4.2.1), which implies that we want to reuse them.

4.2.1. The Matrix

Microsteps Matrix

Microsteps can be arranged into a two-dimensional matrix Figure 4.2. We can plot

[3]See the AST Meta Model section for property objects of a node.

the Microstep operation on the two axes: target language elements, and language constructs per language element. Language elements plotted as columns are, for instance: methods (or constructor for that matter), classes (or interfaces) and fields. The rows of the matrix are language syntax constructs which are characteristics of the language elements targeted to be transformed, like: declarations, naming, signatures, modifiers, initializers and references.

For example, the Java language supports the notion of a method declaration, a field (or var) declaration or a class declaration. Here, the language elements are: a method or a class or a field. The language construct we address here is the declaration of an element. Declaring an element means adding an element to the AST tree and the reciprocal (deleting the element) undoes the method declaration.

The resulting matrix has been optimized for our refactoring purposes. For example, we leave out transformations like adding a local variable or adding comments and code annotation. The corresponding matrix setup is not exhaustive. For example, we do not cover Java module and package scoping.

The number of involved Microsteps differ per refactoring. Take for instance, the Change Function Declaration refactoring regarding the simple Mechanics variant. Here, we conduct a search and replace of the method name with a new name for the method, effectively performing the CM (Change Method) Microstep. For the migration Mechanics variant however, almost all the Method-related Microsteps listed in the matrix get involved, as we will see in later examples.

4.2.2. Mapping Actions on the Matrix

As part of our analysis, to determine actual risks involved when refactoring, we focus on the action typed directives. Refactoring actions are the directives that do transform the source code. In reality, however, this also implies that we may alter the expected behavior of the affected code.

Modifying code goes hand in hand with introducing risks, so we need to investigate which risks can be expected for those actions belonging to a particular refactoring. To investigate on the actual dangers, we follow the strategy of diagnosing the code beforehand, that is, before triggering any possibly dangerous refactoring actions.

If we have identified all required actions for a refactoring, as described in What-If develop recipe (subsection 5.2.2), we will associate them with the Microsteps The Matrix (subsection 4.2.1).

4.2.3. Matrix in relation to Risks

We want to map our Microsteps to potential risks as the source to our What-Ifs as discussed in Source code Diagnostics (chapter 5). The potential risk per refactoring mechanic step can be determined by watching for potential dangers per Microstep we encounter. However, the context of the subjected code also determines possible threats to deal with. As said earlier, the granularity of the Microstep is chosen to

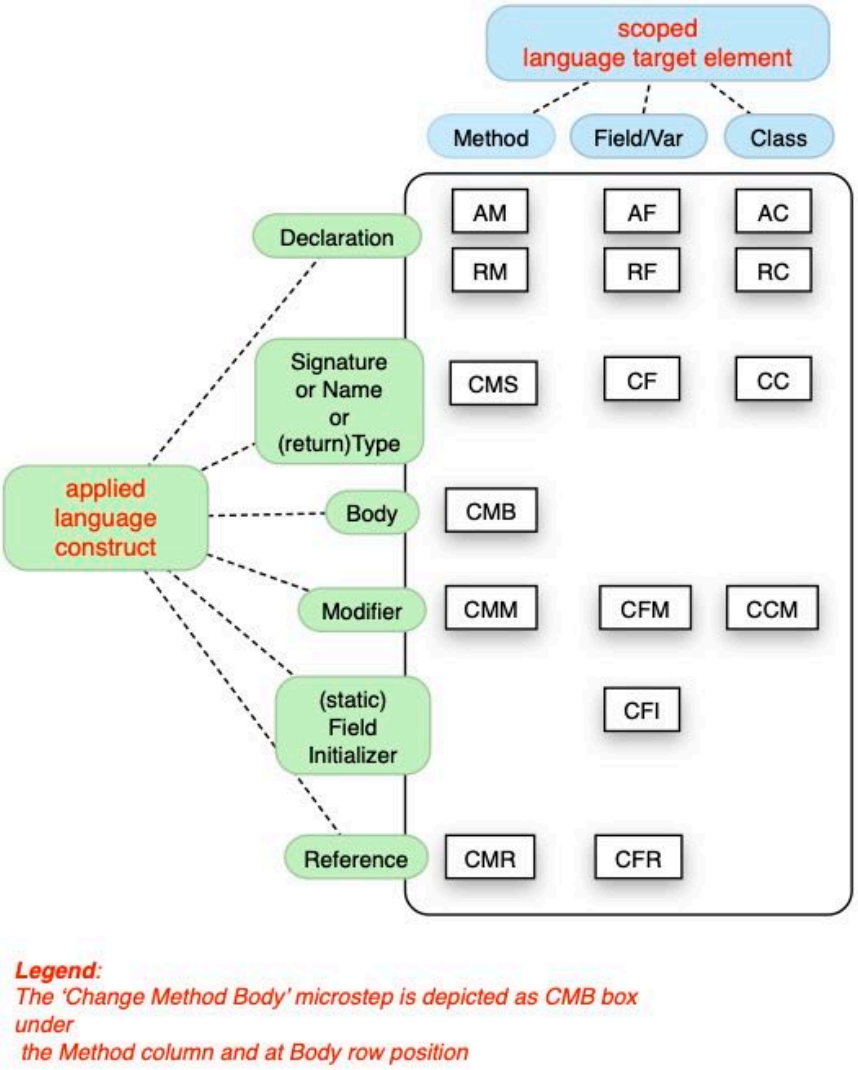


Figure 4.2: Microstep matrix

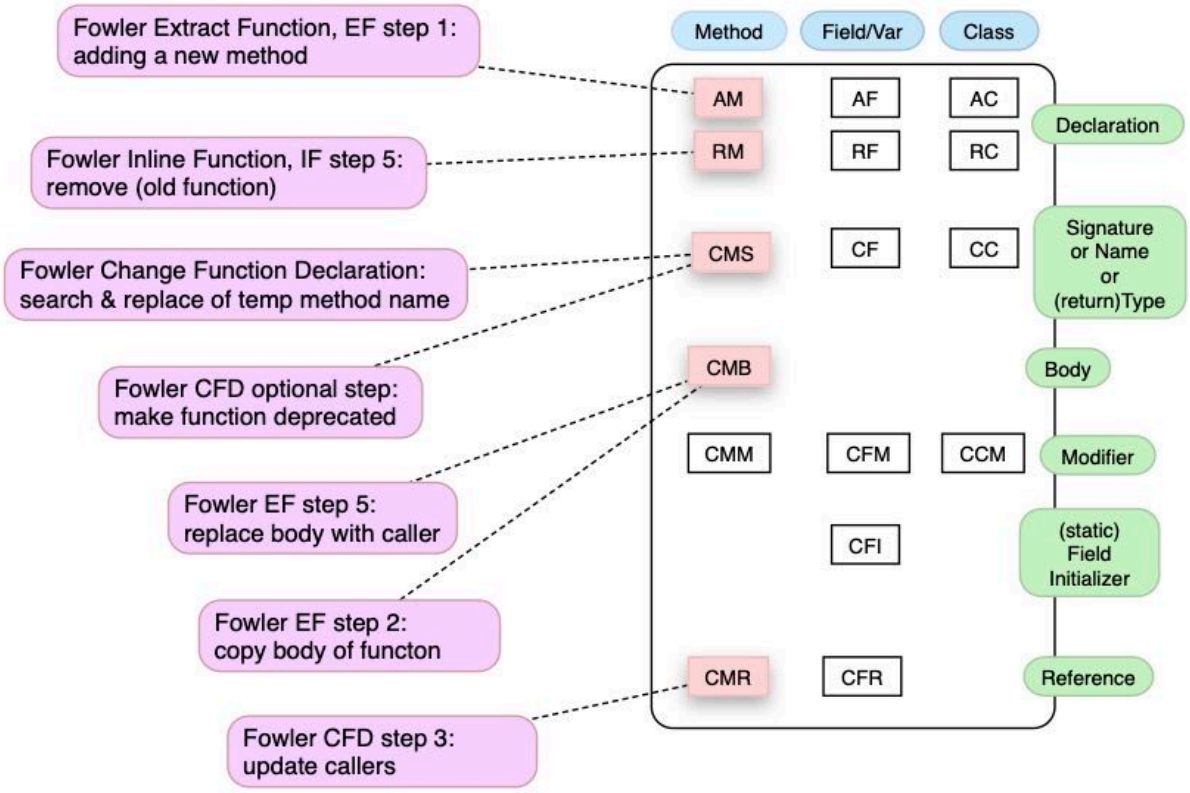


Figure 4.3: Change Function Declaration to Microsteps

balance between a still maintainable reoccurring set of Microsteps covered by the matrix and all kinds of risks one might encounter during the refactoring execution.

The following factors determine or influence the manifestation of potential risks for a refactoring:

- The execution of a Microstep resulting in transformations on code. Each operation is either adding, removing or changing the AST
- The target being operated on, affecting the scope of operation
- The language construct in regard to actual code context is subject to the refactoring.

Notice that potential risks themselves are all possible theoretical risks, regardless of the actual context.

We have put the above ingredients into a Matrix as source for potential risks. These potential risks can act as a foundation to derive our What-Ifs. In tables Potential Risks #1 for Microsteps (Table 4.2) and Potential Risks #2 for Microsteps (Table 4.3), we mention programming concepts like: overriding, overloading, shadowing, obscuring and hiding as a common source of errors. Improper use of these fundamental concepts is dangerous to the functionality of our code.

4.3. Matrix properties

4.3.1. Scoped elements and language constructs

The number of relevant Microsteps, is (not surprisingly) limited in relation to the number of language concepts. Two properties of the programming language play a significant role when refactoring: *java language constructs* and *scoped java elements* (targeted for refactoring). Besides the element under investigation, scope is also determined by access modifiers.

Column properties

In Figure 4.4 we see the typical Java scope subdivision for packages that can group multiple source files as classes (per source file, one public class). The picture illustrates that though we can have multiple packages with multiple classes, the project's scope is even more comprehensive and it can contain multiple packages. Classes on their own can contain multiple methods (and fields), but statements reside only in the method's body. For now, let us focus only on classes and methods to address the property of targeted elements.

With the notion that classes and methods as scoped elements, how can we deduce Microsteps? We should figure out how classes and methods are intertwined. For this, we got inspired by the mechanisms of how ASTs are crafted. Based on an article how to construct an AST from Tom Mens [Mens et al., 2005], we distilled the exposed interactions patterns between classes and methods, as depicted in Figure 4.5.

Methods reside in a class, as we can see by the method membership relation. However, methods are also associated in different ways with classes. If we look at a method, the method is identified by a method name and by its signature and parameters. The established parameters can refer to a class as an input argument (right

Table 4.2: Potential Risks #1 for Microsteps

Language construct	Operation	Target	Potential Risk examples (non exhaustive)
Declaration	Add	Method	Compile errors and warnings, because of: Name clashing. Behavior preservation break, if introduce unwanted overloading when signature differs, interfere with overriding when adding method as part of the inheritance chain
Declaration	Remove	Method	Compile error with orphaned references (statically determined). Method deletion may break behavior preservation in case of inheritance or with overloading. The whole method body will be deleted as well.
Declaration	Add	Class	Same as with methods, the parser will notice name clashes within the same package scope. However, adding as an inline class is allowed (if not already added previously) with the same name as parent class.
Declaration	Remove	Class	Removing a class also removes all containing fields and methods along with their bodies. This aspect may lead to compile errors. API Classes should remain untouched or intact as deprecated
Declaration	Add	Field	Member variables (declared outside a method body) may get shadowed if there exists a local variable (declared inside a method body) with the same name. Clever parsers may notice this with a warning as this might lead to unexpected behavior breaks. Also risky is a field with same name but declared in a subclass that hides the parent field declaration
Declaration	Add	Var	Vars in local scope might overshadow fields (as explained above) when having the same name. A local variable can obscure a class or reserved names with the same name (for example, the use of System as a local String var along with System.out) failing to compile
Declaration	Remove	Field /Var	Deliberate removal of used vars as part of the refactoring process generates only temporary acceptable compile errors to be fixed

Table 4.3: Potential Risks #2 for Microsteps

Language construct	Operation	Target	Potential Risk examples (non exhaustive)
Signature	Change	Method name	The effect of renaming the method name introduces the same risks as when adding or removing a method
Signature	Change	Params	A static method with the same name and signature in parent and subclass, known as method hiding, may break behavior preservation. The signature is also relevant for the overloading mechanism. Not only the number and order of parameters should be taken in consideration also inheritance plays a role for identical parameter position method counterparts
Return type	Change		With method overriding the return type may differ only in sub-type, fortunately the compiler will detect this situation, but it may lead to alteration of functionality
Body statements	Change		Diverting program execution by inserting or deleting flow control statements, may affect the behavior is such a way that it breaks functionality and/or test results
Modifiers			Access modifiers alter visibility of the service. Removal of public keyword is not allowed when method is API-function. Unintentional change of non-access modifiers can lead to strange behavior of code execution
Initializer			Risk for semantical errors as illustrated at Risk factors (section 3.2)
Reference			Dynamic binding problems, wrong class referrals, are out of scope

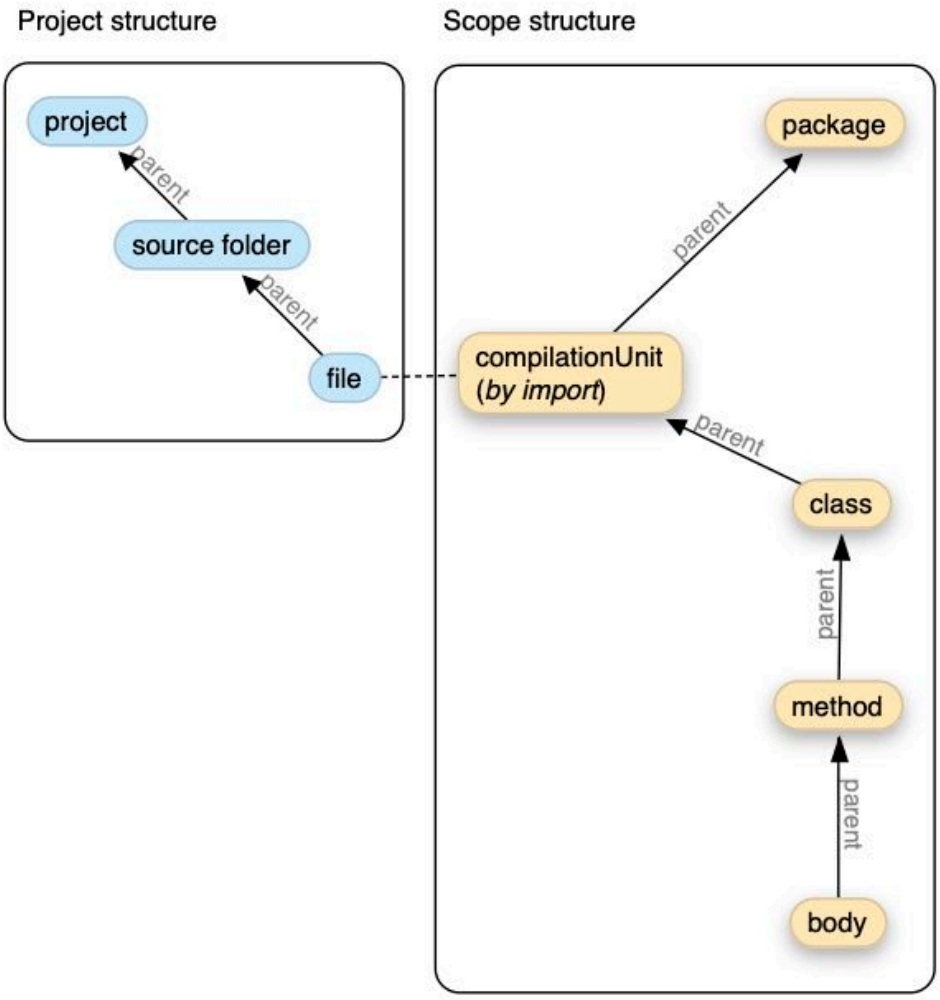


Figure 4.4: Java scope and project structure

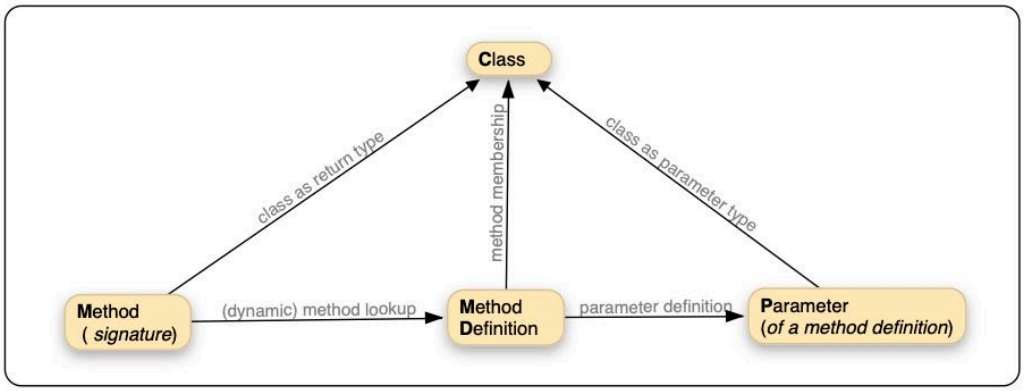


Figure 4.5: Method and Class relationship

arrow between parameter and class), but the class may be the return type of a method call.

Row properties
The row properties of the Matrix represent *language constructs*. If we dive into the Fowler refactor catalog, reasonably many of them (at least the most relevant discussed here) can be covered by the following language constructions to be contained in the Matrix: declarations, method signature/name/return value changes, method body transformations, method modifier changes, (static) field initializer changes, and method lookup changes.

4.3.2. Method-scoped Microstep appliance

As we know already, language constructs are an essential aspect to define Microsteps for the matrix. When we want to apply transformations on the target elements: method, fields, or classes, we must be aware of the syntax to derive the detectors to examine the code for potential dangers when applying selective refactoring.

Method and Fields syntax

The Java language issues syntax rules to obey for every detector. As we know, a Method Figure 4.6 must reside within a Class but cannot contain Fields. Both Methods and Fields are scoped by a Class (or Interface for that matter, since an Interface is a special case of Class). Notice that the AM (Add Method) Microstep should reference an existing parent class.

Fields
A Field variable is, likewise a method, a member of a Class (data members are declared outside any Method or constructor but inside a Class block). Fields are therefore variables of a Class (being either instance variables or static variables). Java only supports global variables because it acts as Class variable, utilizing the static keyword indicator. A variable is a name given to a memory location and variables have a

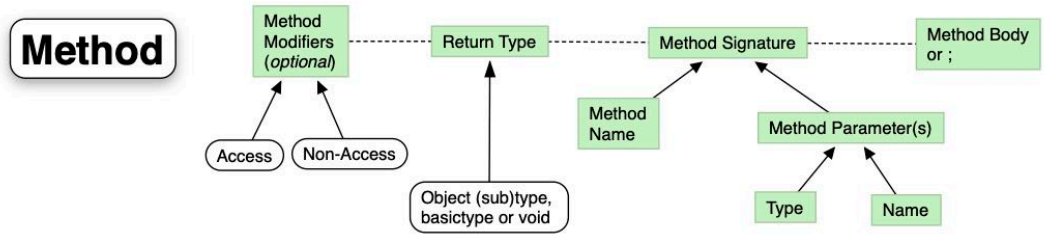


Figure 4.6: Method Appliance

type and a scope. This means that the AF (Add Field) Microstep also must have a reference to an existing class as input argument and a name and type to identify the Field: AF(class, field)

Methods

Java methods can contain statements and variables. These so-called local variables are variables declared within a method, constructor, or a specific block of statements. A method body can be regarded as a special case of a statement block; it encloses possible statements with method scope. This implies that a new method will be constructed by an AM (Add Method) Microstep followed by a CMB (Change Method Body) Microstep to add statements. The CMB is also used to alter the source code on statement level. As input, the CMB contains a list of statements and an insertion point. If the inserted statement is the empty statement, effectively, a delete is in place. Execution of statements is in sequential list order.

Summary for Microsteps, operating on the Method level:

- AM: Add Method:
- RM: Remove Method:
- CMS: Change Method Signature
- CMB: Change Method Body
- CMM: Change Method Modifier:

Summary for Microsteps, operating on the Field level:

- AF: Add Field
- RF: Remove Field
- CFM: Change Field Modifier
- CFI: Change Field

- CFR: Change Field Reference

The AST construction for the above Method syntax Figure 4.6 is covered by Meta Model-level example (subsubsection 6.2.1).

Method Call syntax

Method references

Statements within the method body should be modified with the CMB Microstep (Change Method Body, as seen at Method and Fields syntax (subsubsection 4.3.2)). An exception is made for the so-called receiver referring statements, like method calls and field references. These kinds of statement expressions contain the Java dot operator (for instance: *receiverObject.field*). They are excluded (and are Microsteps on their own) because, in many refactorings, Fowler separates Mechanics steps for dealing with the method and dealing with the callers to the method.

As mentioned, the CMB Microstep is too generic when it comes to updating object references. For example, the Rename Method refactoring incorporates a CMS (Change Method Signature) and a CMR (Change Method Reference) for all affected callers based on the new name. The same applies to the CFR (Change Field Reference) Microstep, changing to the new field name. Both CMR and CFR instead change the lefthand side of the dot regarding the ReceiverObject identification Figure 4.7.

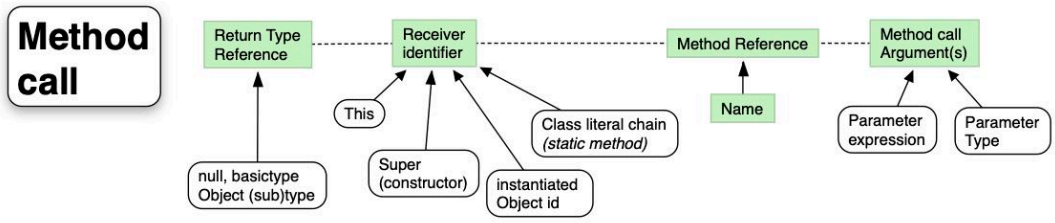


Figure 4.7: Method Call Appliance

Method binding

When refactoring methods, issuing CMR Microsteps, we must distinguish between static binding and dynamic binding. With static binding, in case of overloading, we can infer the object type at compile-time. However, when assuming overriding, we encounter issues the AST cannot resolve. With static binding, the receiver object can be determined and compared to the involving method's object. Though in the case of dynamic binding, the AST can only resolve to the receiver object and type known at compile time but not (easily known) at runtime object[4] unless deducted by human intervention. Fowler warns about refactoring polymorphic methods!

[4]Image for the refactoring, with dynamic binding, potentially, we need to check for all the (sub/super) classes the receiver object can point to.

Class syntax

A Class or an Interface Figure 4.8 can be either a top-level Class (defined in a CompilationUnit scope by Package) or defined within another Class. This nested class construct is the concept of Inner Classes. If a particular class has the static modifier, the class can only reside directly under or at the same level as the top-level class (and can only contain static members). A specialized class, the anonymous class, can be defined within a method scope Figure 4.6.

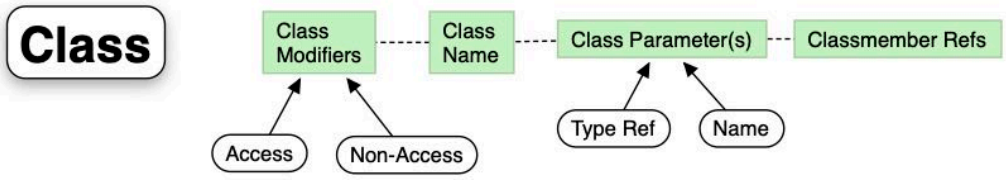


Figure 4.8: Class Appliance

The AC (Add Class) Microstep is needed to create a new class within the context of a package.
Creating an anonymous class (classname omitted and declared within method scope) is achieved under the supervision of the CMB Microstep Method and Fields syntax (subsubsection 4.3.2).

Summary for Microsteps operating on the method level:

- AC: Add Class:
- RC: Remove Class:
- CC: Change Class:
- CCM: Change Class Modifier:

5

Source code Diagnostics

If we want to reason about the presence of potential dangers, we need to be aware of the structure and state of code. Potential dangers can turn into real actual dangers when we exercise the refactoring.

In this chapter, we want to work out the idea of Risk-based Refactoring (chapter 3), taking on the machinery of code diagnostics. Code diagnostics is the vehicle to harvest facts about the current state of code.

5.1. Detectors

With the aid of detectors, we can determine the state of code. From the inside, detectors can be considered as building blocks of coded logic, with the core functionality of scanning the code context for specific characteristics.

5.1.1. Detector concept

In this section, we will talk about these detector building blocks, how to compose a detector, and the different detectors.

Detector types

We differentiate between detector types based on their purpose. We can distinguish between three types of detectors; detectors in general and the specialized detectors: Selector and What-If type.

Detectors types:

1. Selector specific; pertained to the subject to be refactored. See Which code? (subsection 1.1.3)
2. Mainstream (generic) detector; general-purpose code scanning vehicle
3. What-If specific; as discussed in upcoming What-Ifs section

The Mainstream typed detector is the common nominator type. If we would regard detectors in an object-oriented manner, the Mainstream detector can be considered the superclass for the others. As depicted at Table 5.6.

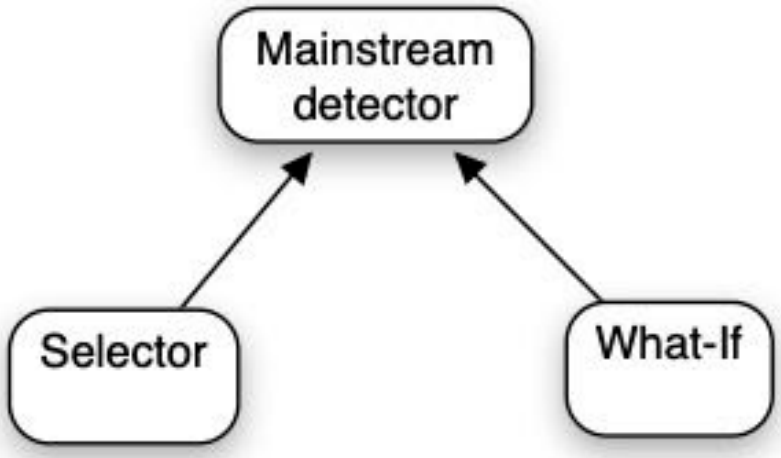


Figure 5.1: Detector inheritance

Requirements

Requirements for detectors:

- From a design point of view, the detector should comply with the Single Responsibility Principle, with the expected benefit that small tasks gain better support for reuse.
- Detectors should entail meaningful outbound Data format towards either other detectors or What-Ifs.

Concerning the What-If (and the Verdict engine (subsection 5.3.2) incorporated into a What-If, as discussed in Verdict Idea (section 5.3)), the What-If requires meaningful data from at least one detector to draw conclusive advice. Conversely, for What-Ifs, it is not necessary to pass on data because the advice presentation is an end situation.

Regarding the setup of both Detector or What-If, their construction should preferably honor the Separation of Concerns Principle. The What-If should therefore abstract itself from obtaining the state of code directly. The task of querying the AST should be delegated to our detectors. (An expected benefit is that conceptually related languages can be supported by slightly adapting the detectors, without changing the What-If.)

Catalog of detectors

It is good practice to assemble a collection of (utility) detectors to generate unbiased sets of data, like, for instance, a list of all methods or all classes from the project. Elaborated on data generation, it is a relatively small step to narrow the scope, or to filter the list based on name or other criteria, further specializing the output required for the What-If.

Remark: *We could consider building library functions for dealing with the AST directly.*

Detection on demand

The ideal situation would be to continuously monitor the state of the source code after each user edit. Notwithstanding here, we run into a couple of problems that made us postpone the optional research question RQ4 to future work.

- Firstly, there must be a well-defined demarcation of performing code transformations. For example, a copy-paste action should not be seen separately but as one atomic action. So it makes more sense to do code diagnostics only after the paste action has been completed
- Secondly, to track progression, you need to set up mechanisms to enable which phase of the refactoring the user is currently doing or wants to do. Every single action needs to be traced.

5.1.2. Detector internals

Below you see a visualization of the detector internals detector concept (Figure 5.2). Every detector consists of building block components. These containing blocks operate on data assembled from the code under investigation. But what is this assembled data? In these sections, we explain the following:

- Internal Detector Building Blocks concept (subsubsection 5.1.3)
- Format of input/output streams; Data source (subsubsection 5.1.2), Data format (subsubsection 5.1.2), List size (subsubsection 5.1.2)
- Detector interaction concept (subsection 5.1.6)
- Blackbox notation (subsection 5.1.4) of a detector

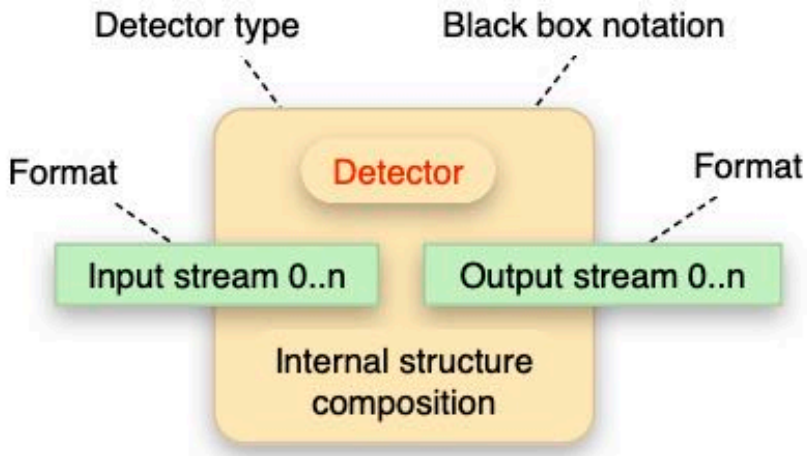


Figure 5.2: Detector concept

Data source

Assembled data is data gathered from multiple possible sources, by any combination, but at least based on one of:

- internally called AST query operations, containing AST node references as a result of the query.
- data retrieved from other detectors or What-Ifs through streaming channels. By default, each stream represents one list of AST node references.

- (pre-configured) internal data.

  What-Ifs, for example, create decision tables. See Verdict Idea (section 5.3) about administering numerical weight values.

Data format

Regular usage

We propose that both input streams and output streams are presented as a list (of objects) for passing code state between detectors. As mentioned, these objects comprise AST node references and represent a language element as elaborated at Conceptualizing the AST (section 6.2), including source code location.

Customized appliance

Detectors can agree, for example, to pass other kinds of data than AST object references between each other, and this data would be only meaningful between these detectors. Imagine a detector that will provide the number of parameters per method; by contract, the acquiring detector knows about the meaning of the elements in the list.

List size

Empty lists are allowed. Image, we query for static methods in a class. Without static methods, the resulting list will be an empty. How the detector interprets the list size is its own business. By convention, an empty list represents the logical FALSE, conversely, when the size of the list equals one or larger, the detector or What-If can interpret this as the TRUE value.

5.1.3. Detector compositions

Each detector is made up of compositions of logic, and in this section, we will address its constituents.

Detector composition example

The best way to explain detector compositions is by example. Image, we want to find all classes for methods having the same signature as the refactoring subject, all within the scope of the same package. The task for our example detector is to deliver the matching methods to extract the resulting classes. Applicability: in the overall Rename Method refactoring scheme, we want to inform the user that, in the case of involving abstract methods, also all those derived method declarations need to be renamed.

According to the Single Responsibility Principle, our detector needs to resolve if the source code within the scope of the same package contains methods with the same name as our target renaming method.

In this example building block (Figure 5.3), we will zoom in on our cozy detector named *detb0* (stands for DETector Building-block tree node-number *det0b*), with the Blackbox notation (subsection 5.1.4): *det0b:same-methods?@enclosing-package*
As you can see, you can already guess, based on the naming, that the detector should find the same methods within the range of the enclosing package and those same methods are similar in signature.

Scoping
Our example detector works in cooperation with other detectors to cope with this abstract method scenario. Here the sole task for our detector, with the alias name *det1* for *det1:classes?@same-methods*, can be described as "find all classes for methods with the same signature within packaging scope".

An essential aspect of a detector calling another detector is that it builds upon the 'knowledge' of the callee detector. The *det0b* detector already limits the scope to the package in which the method resides. This joint effort of detectors makes the Detector chaining concept (subsection 5.1.5) possible.

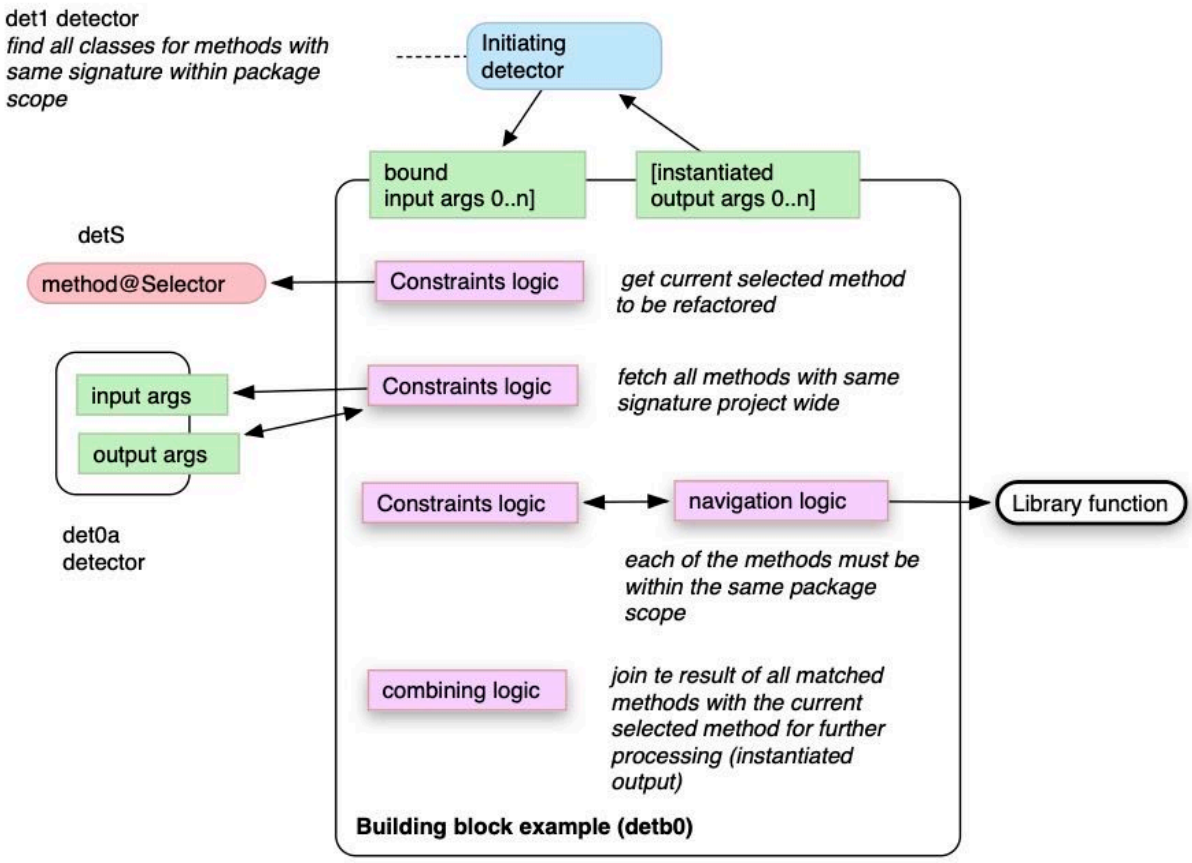


Figure 5.3: detector composition example

Logic
Let us dissect this detector composition example Figure 5.3 part-wise. All parts are based on the underlying logic types described in Detector Building Blocks concept

(subsubsection 5.1.3).

Constraints logic:* "get currently selected method to be refactored"*; Because of Separation of Concerns, we delegated the data about the current method selection to the dedicated* detS *selector. Fetching the output stream from another refactor is an example of constraints logic. The delivered characteristics about the method to be renamed is input for further processing.*

Constraints logic: "fetch all methods with same signature project wide"*; This step's task is to fetch methods by invoking another detector:* det0a:same-methods?@all*, meaning: fetching all methods with the same signature project wide, it is suffice to say that this detector does the heavy lifting because it assembles a big part processing data; that is, a list of same methods from the whole source base.*

Constraints + Navigation logic: "each of the methods … package scope"*; The core functionality for this detector* detb0 *is, independently without aid from other detectors, processing the assembled data (consisting of a list of all methods from* det0a *and the targeted method from* detS*) to filter all those methods that do not belong to the enclosing package. However, to determine the enclosing package, we need first to determine the method's enclosing class.*
Concerning the navigation logic, operations like enclosing objects, containment and inheritance; these are typical candidates to be exposed as public library entries.

Combining logic: "join the result …"*; The final step for our example detector is to join the other matched methods, including the targeted method itself. Typical use of applying combining logic is to merge multiple input streams into a single stream. Because we work with lists as Data format (subsubsection 5.1.2), we can combine our refactoring subject and matched methods with the union operator.

Detector example in Prolog

The detector composition implementation results in following code listed. The language used for our prototype is Prolog. The code is almost self-explanatory, which makes it easy for prototype development.

```
1 /* det0_matching_methods(-MethodList, ?PackageId) */
2 det0_matching_methods(MethodList, PackageId) :-
3   selector_method(InputMethod, _, PackageId),
4   det0_matching_methods(Methods),
5   findall( Method, (member(Method,Methods),
6               encl_package(Method, PackageId)),MethodsInPackage),
7   AllMethods = [InputMethod|MethodsInPackage],
8   sort(AllMethods, MethodList).
```

Listing 5.1: Detector composition example

Parameter handling

From the standpoint of a function, our detector acts accordingly as a function with input and output parameters. For Prolog, we have to mimic the mechanism of output variables; here, we have the notion of bound or free variables. Passing input arguments, in the sense that they have been assigned a value (c.q. unified), are the ones

we call bound variables. For output variables, we expect, as post-condition, that they have been instantiated (likewise unified, bound, being assigned a value) after the call to the function. The pre-condition of a variable may be either bound already (argument type both input+output and can act as optional argument) or free at call time. In general, after the function call, we expect all arguments are non-variable anymore; in other words, they contain a value.

In the example detector listing above, the methodList argument, denoted by a '-' prefix, indicates that the parameter must be free when passed (not assigned a value at function entry). It represents an output argument. To facilitate optional arguments The PackageId, prefixed by '?' can either be bound or free. The '+'prefix indicates that the argument must be bound at call time.

Detector Building Blocks concept

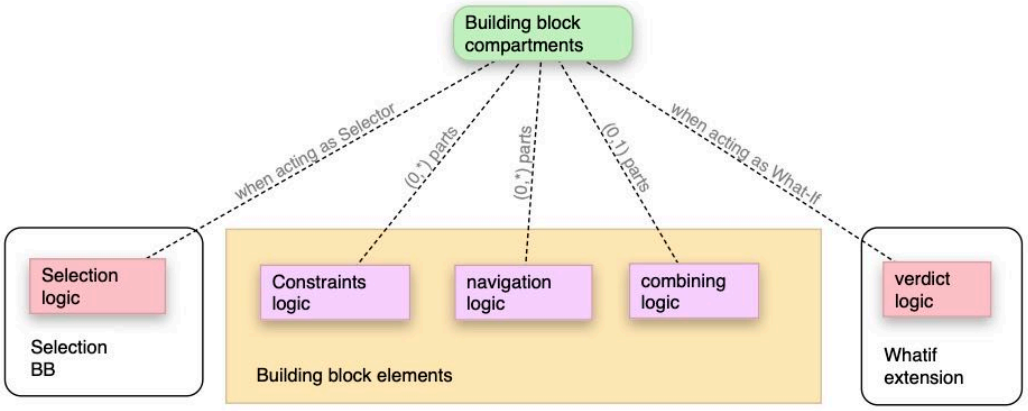


Figure 5.4: Detector building blocks

Detectors Figure 5.4 make use of different kinds of processing logic and we call them the building block elements of a detector. In the tables below and accompanied by a Detector composition example (subsubsection 5.1.3) section, we distinguish between logic and which of the detector types make use of them.

Logic to be applied by any detector type:

Specialized detector type What-If should contain:

The Selector detector only contains:

Remarks

Constraints logic, ad (A): Code context properties are typical AST facts that represent language constructs like: methods, classes variables, packages etc., along with modifiers based on actual code context. Such a property[1] written as an expression may be

[1]Patrick the Beer [de Beer, 2019] refers to these properties as code constructs present in the code context. The abbreviation he uses for Code Context Properties is CCP. Any detector implementing a CCP is named CCPD (Code Context Property Detector).

Table 5.1: Detector building block base logic

Logic Type	#parts	Application
constraints logic	(0-n)	(A) code context property expressions (see explanation at remarks), mapping and/or filtering functionality, invocation of (other) building blocks to enable chaining of detectors
Navigation logic	(0..n)	(B) inheritance, containment, nesting
combining logic	(0–1)	(C) intersection, union, disjunction

Table 5.2: Detector building block primary What-If logic

Logic Type	#parts	Application
Verdict logic	1	(D) IF <verdict matching conditions> THEN <output messaging>

combined with other property expressions by means of Boolean AND/OR/NOT logic combinations.

A significant constraints logic feature is that of filtering and can be applied to input or detector output. Filtering on the input is scope restricting and filtering on the output is restricting on the subject. Subject and Scope are the black box indicators for a detector with notation $< subject > @ < scope >$ as described in Blackbox notation (subsection 5.1.4). A concrete example of filtering and other logic-type implementations can be found at Detector chaining concept (subsection 5.1.5).

Navigation logic, ad (B): OO-related navigation logic; for traversing the AST concerning super/subclasses, direct or via transitive child/parent edges. Containment logic for getting the enclosing class or method or block depending on selection. Nesting traversing facility when to deal nested structures like inner classes.

Combining logic, ad (C): Combining logic is supplying the operators for assembling the result from multiple other detector sources. For instance, if detector 1 matches class A and detector 2 matches class B. The union yields [A,B]. In Detector Building Blocks concept (subsubsection 5.1.3), we see the union operator in action.

Verdict logic, ad (D): The output from Verdict logic must comprise at least a list-size > 0 number of matches. Every match contains an output message, including the appropriate source location.

Table 5.3: Detector building block selector logic

Logic Type	#parts	Application
Selection logic	1	(E) Selected code by the user

Selection logic, ad (E): For the Selector detector, the selected code by the user is looked-up for in the AST. If, for example, the selected subject to be refactored is a method name, the AST node representing the method will be returned by this detector. Every AST node has internal knowledge about the code structure span of control, including the source code locations.

5.1.4. Blackbox notation

The Building Block structure reveals details about the inner workings per detector. However, most of the time, we only want to know what the detector is doing in regard to its functionality, but not how it is done, since inner working details belong to the implementation realm. Within this thesis, to specify our detectors, we will use both the black box version and white box version formats.

The idea of introducing a shorthand notation was an initiative born at one of our Refactor Buddy working group sessions to help us view the detector as a black box. The information in the notation language should supply just enough detail to know what the task of the detector is.

Demands

We drafted the following demands as a starting situation:

- The notation should be compliant with the expected functionality for that detector,
- Because a detector can have both input and output stream channels, the notation should reflect this aspect,
- Formulas or statements used, preferably, should not be too lengthy but relatively concise enough to express their intention. As long as the intention is clear, we do not oppose expressions in natural language form. However, in future versions, we can imagine using a notation that resembles more of an XPath kind of construct because this way, we can improve on standardization and have richer expression possibilities,
- Use the plural form to indicate a list of elements. The question mark indicates the subject(s) we are interested in when calling the detector. The alias format (below) is introduced to avoid repetition and easily refer to other detectors to support detector chaining. For our example detector at Detector example in Prolog (subsubsection 5.1.3), we could express the *det1* detector as *det1:classes?@det0b*

Format

The notation of one detector should adhere to this generic format:
< subject > @ < scope >

Notation

The *< subject >* part can be anything like processing constraints, description of the functional result, or XPath alike expressions. If we have more than one output stream,

then the output streams are grouped by a comma.

The at sign '@' separates the subject from scope, or the detector's output from input.

The *< Scope >* part is, in fact, determined by the depending input detectors. Hence, you may also use the notion equivalent form of:
< alias name > := < processing-constraints > @ < input sources >

Again, no restrictions are enforced!

Detector aliasing
In case of supporting detect chaining, any notation per detector should be proceeded by an explicit name as in:
< detector name > := < subject > @ < scope >

To express depending detectors, we can refer to a comma-separated list of alias names. However, aliasing is not a requirement though. You can just decide to repeat the complete notation of the referring detector(s). As you can see in the example, *classes? @ [same-signature-methods @ package]* is more difficult to grasp than to write down *d1:classes?d0* based on the notion that we already have defined *d0.*
We make heavy use of aliasing when we construct our Detector chaining concept (subsection 5.1.5) trees in the next section.

Examples
Again, the notation expressions style has not yet been formalized and left for future work. Our goal is to describe the bits and pieces concise but detailed, as long as it is clear what the purpose or intentions are for that detector. Example black-box notations: Table 5.4.

Table 5.4: Blackbox notation examples

Detector query	Black box notation
method m from class A superclasses of this method m also containing same signature method m	Detector1: method-m?@class-A Detector2: superclasses?Detector1 Detector3: method-signature?=detector1.signature@superclasses

In Detector chaining concept (subsection 5.1.5) section, we see further examples of black box notations. Notice here the prominent role of aliasing detectors with a lower position in the chain. The black box notation in regard to the AST can be seen furthermore at the sections AST representation (subsection 6.2.2) and EF-refactoring demo 2 (subsection 6.3.2).

5.1.5. Detector chaining concept
In section Detector composition example (subsubsection 5.1.3), we introduced the inner details of one example detector (named *det0b*). Typically, this white box approach

is not handy when designing and developing a fully-fledged What-If implementation. In this example, the What-If scenario is one of the series What-Ifs described in the section Based on What-Ifs (subsubsection 5.1.7), namely, the Rename Method of an abstract method

In this case, refactoring practitioners should be warned about the implications that, if they want to rename a method declared abstract (our Refactoring Subject), the consequence is that related classes will also be affected and need to be renamed. If we know the affected classes, we can pinpoint the individual methods, but this is a task for another scenario (in the spirit of Separation of Concerns).

We will apply this What-If on abstract classes presented earlier in Complex refactoring (subsection 1.2.3) in figure Figure 1.1.

Depending on other detectors' output, detectors can be depicted as directed graph nodes (within a DAG), with an edge representing the dependency. In figure Figure 5.5, we have drawn green nodes representing those detectors (denoted as *det<0–5>*). To distinguish between What-If and detector typed nodes, the What-If node has been given the blue background color (denoted as '*whatif*'). The red boxed node is the selector detector type (denoted as *detS*). This detector is aware of the particular abstract method to be renamed.

The execution point for this example starts at the *det6* What-If, the top-level element of the graph.

As part of our What-If, the verdict engine Reasoning mechanism (subsubsection 5.3.2) needs the results from detectors *det5* and *det3* to determine which classes are candidate to report. The edges between the What-If and these detectors representing the calling dependency between the nodes; What-If calls *det5* and *det3*. We reuse the same instantiated output steam data since det3 will be called twice, indirectly from path [det6, det5, det3] and directly from path [det6, det3]. In the graph, you will notice reuse of detectors, like *det1*.

Because detector reuse is revealed already within a single What-If case, in suiting cases for other refactorings, we can assume that reuse is not an exception. Formal proof for this is out of scope for this thesis and left for further study assignments

An indispensable feature for detectors, as demonstrated, is filtering functionality constraints logic, described in Detector Building Blocks concept (subsubsection 5.1.3). For instance, the detectors *det0b* is filtering on scope (all to package scope), and *det3* is restricting on subject (superclasses to abstract superclasses)

Another feature is traversing the internal AST data by navigational logic to obtain the transitive closure[2] of class to superclasses, as witnessed by *det2*. We also witness the power of combinational logic embodied at detector *det3*, which merges matching

[2]Here, transitive closure implies including not only the direct parent or child but also their parents or children as well, all the way up or down to top–level or leaf classes

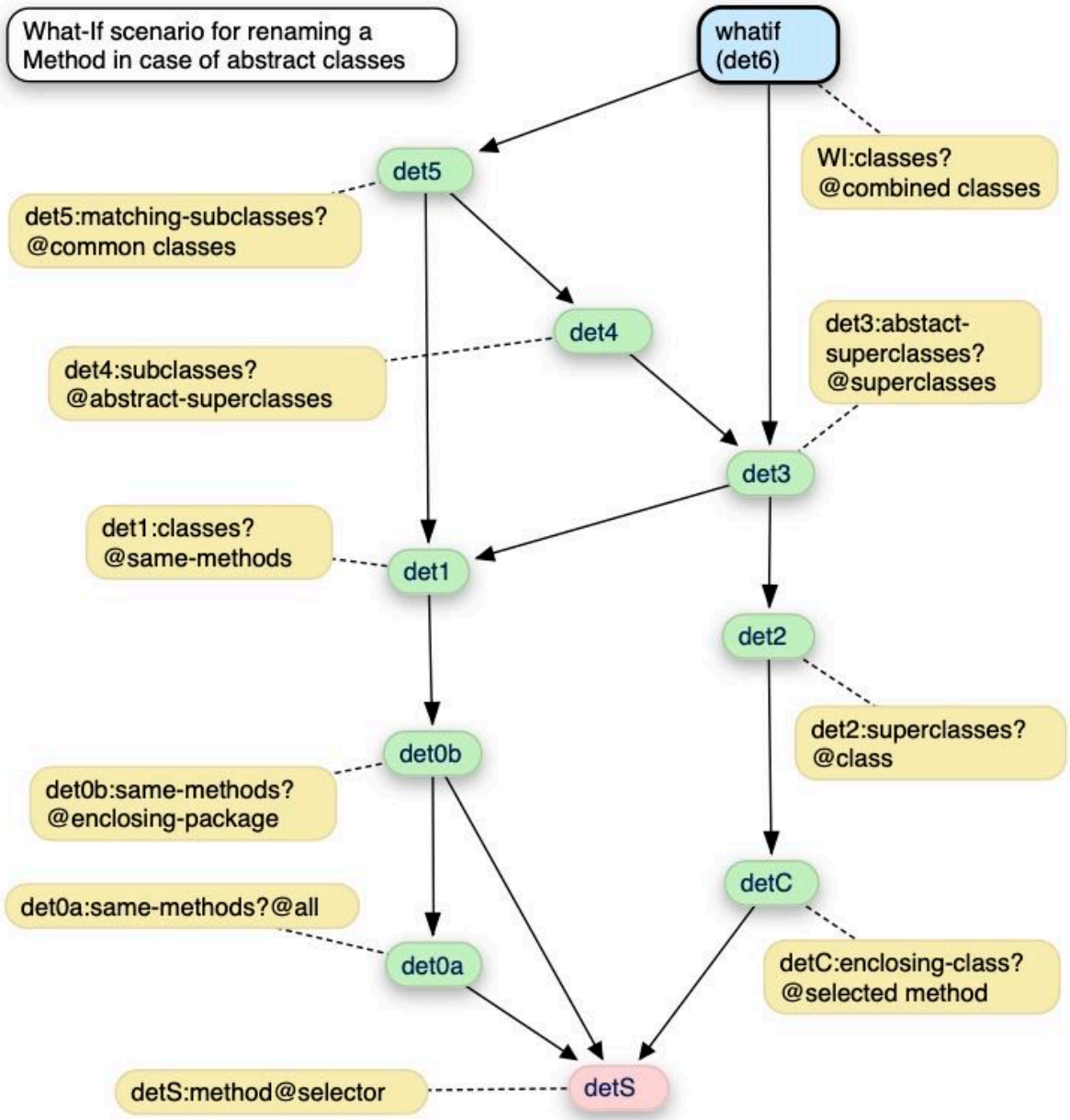


Figure 5.5: Detector chaining example

superclasses with those classes containing the affected method similar to our Refactoring Subject.

The What-If detector chaining example presented in figure Figure 1.1 has been captioned in the following table Table 5.5. Besides a summary of the logic constellation, per detector, as discussed at Detector Building Blocks concept (subsubsection 5.1.3), we introduce two more concepts here. The Layering model (subsubsection 5.1.6)and Detector sub-typing (subsubsection 5.1.6) items (first column A-E indication) are discussed in the section Detector interaction concept (subsection 5.1.6).

Table 5.5: abstract class renaming What-If

Detector# + subtype	Applied Logic	Layer
Det6 (E)	Verdict by combining	What-If
Det5 (C)	Combining	Detectors
Det4 (C)	Navigation and Constraints	
Det3 (C)	Constraints and Combining	
Det2 (C)	Constraints	
Det1 (C)	Navigation	
Det0b (D)	Navigation, Constraints and Combining	Utility detectors
Det0a (B)	Constraints	
DetS (A)	Method selecting	Selector

5.1.6. Detector interaction concept

We have divided the detectors into types based on their purpose and their mutual interaction. Based on input/output channels, we can differentiate on streaming characteristics. This kind of division enables us to create interaction between detectors based on layers of detectors. The Detector chaining concept (subsection 5.1.5) (example) has been designed according to this blueprint layering.

Layering model

The following image Figure 5.6 is our blueprint guideline for detector interaction in general.
Detector layering is an important concept to be applied when designing the detector chains.

The most crucial pattern for detector chaining is detectors invoking other detectors. What-Ifs can invoke other What-Ifs or detectors; detectors themselves can call other detectors or the Selector specific detector. What-Ifs are the most AST agnostic; their verdict is based on identified dangers operating on a higher abstraction level, independent of AST construction. A Selector detector, therefore, does not get called by What-Ifs directly.

Another pattern to be regarded is the flow, direction, and starting point. The model is a DAG (Directed Acyclic Graph), with the top-level What-If acting as a root node. We can have multiple root What-Ifs needed to cover a Fowler refactoring. For example,

the Rename Method refactoring is supported by two What-If cases, one covering the abstract class-related dangers and another covering interface class-related dangers.

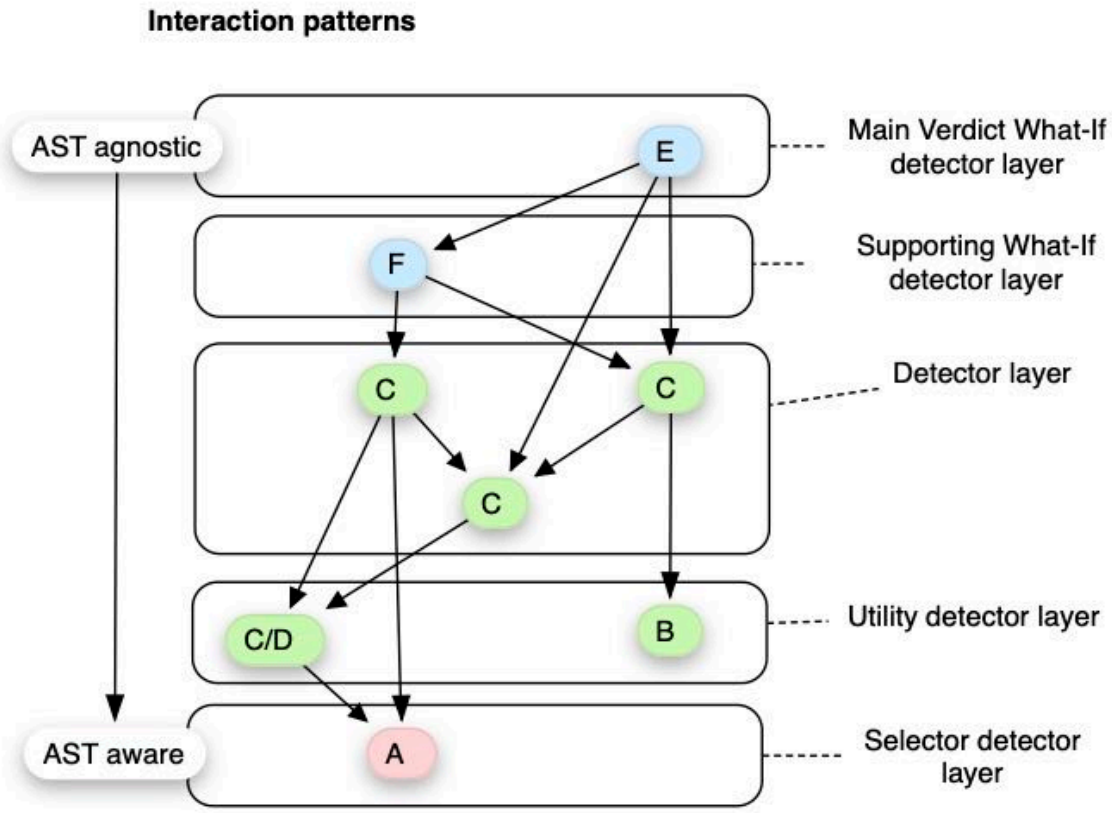


Figure 5.6: Blueprint detector interactions

Let us explain this blueprint interaction illustration layer-wise from the bottom layer up to the top layer.
Color legend:

- The *blue-colored* objects are What-Ifs
- The *green-colored* objects are our Regular or Multi-stream I/O-based detectors
- The *red-colored* object is the Selector specialized detector

The node-naming corresponds with the sub-type designation as discussed in chapter Detector sub-typing (subsubsection 5.1.6).

The Selector detector layer: Only populated by the Selector detector is a specialized detector servicing other detectors. It has a close relationship with the AST because it fetches the selected subject for refactoring directly from the AST.

Detector layer: The bulk of the interaction happens between regular detectors. As you might guess, they together do the heavy-duty work. Some detectors do more than others, which depends on the complexity of logic and if it can either call in the help of other detectors or query the AST by itself.

Utility layer: The main purpose for utility typed detectors is that they are highly utilized in fetching lists of objects from the AST, like, for example, all packages in the project, all classes in a package, all methods in the project, all methods with the same name, etc.

What-If layers: Are the top-level layers (or top-level layer if we have no What-Ifs invoking other What-Ifs). Their purpose is, of course: reasoning about the potential dangers, fairly discussed in Verdict engine (subsection 5.3.2). The supporting What-If detector layer (dedicated to subtype F) exists if we need an arbiter What-If construct (subtype type E).

Detector sub-typing

Besides the type of the detector, we can also sub-type them based on their streaming capabilities. A stream can be inbound or outbound. For example, the Selector detector is outbound only, meaning it does not depend on other detector processing outcomes.

In Layering model (subsubsection 5.1.6), we have seen the detector sub-types mapped onto the interaction layers. The different sub-type detectors make it possible to induce different kinds of interaction patterns. We recognize the following detector sub-types: Table 5.6.

Table 5.6: Detector typing

Detector type	Sub–type
Selector	A) output only
Mainstream	B) output only
Mainstream	C) regular, 1 channel input, 1 channel output
Mainstream	D) multi-streaming, I/O with multiple channels
What-If	E) verdict only (input)
What-If	F) verdict and behaving as detector

I Selector type detector

This type does not have sub-types, this detector is output only by nature.

Subtype A) Selector detector:

- Notation: *< subject > @ selector*, example: method@selector

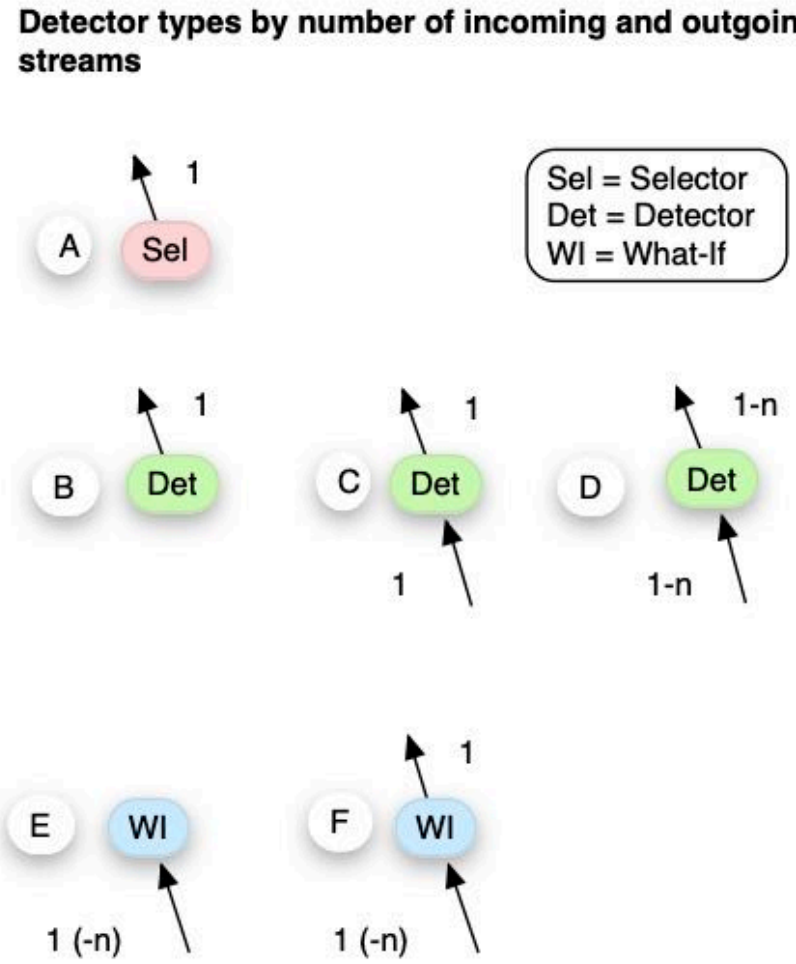


Figure 5.7: Detector Types and Sub-types

- Building block composition: Selection Logic
- Characteristics: has output only and is solely based on the Refactoring Subject, introduced at Which code? (subsection 1.1.3). For example, if renaming a method is the refactoring, the target is a method name. Selection depends on the location within the source code and the range of the selected source code lines. By pointing to a source code location, the user can be explicit which method will be renamed in case of multiple occurrences in the same class, like overloaded methods or the compilation unit.

II Mainstream type detector

This type has sub-types: Output only detector, Regular type detector, Multi-stream type detector.

Subtype B) Output only detector

- Notation: *< subject > @ all* , *< processing constraints > @ all* or simply as *< processing constraints >*
- Building block composition: Constraints Logic mainly, optionally Navigation - and/or Combination Logic
- Characteristics: This detector type has only one output stream. Their task is to gather subjects from the AST directly. Output-only detectors are so-called leaf node detectors (when presented in a graph); their typical use is to provide subject elements, like, for example, all methods within the project because the scope is set to 'all'. Be careful not to think that it cannot call other detectors because this detector has no input stream. Just like a function without arguments, it is still very well capable of invoking other functions.

Subtype C) Regular type detector

- Notation: *< subject > @ < scope >*
- Building block composition: mixed Logic, no Selector or Verdict Logic
- Characteristics: as described by Detector concept (subsection 5.1.1)

Subtype D) Multi-stream type detector

- Notation: *< subject1, subject2 > @ < scope1, scope2 >*
- Building block composition: mixed Logic, no Selector or Verdict Logic
- Characteristics: Besides the same general characteristics from the regular detector type, their typical use is to provide for optional arguments; in the example presented at Detector composition example (subsubsection 5.1.3), we can see the usage of such an optional steering parameter to control the scope.

III What-If type detector

This type has sub-types: verdict only, or verdict and passing on verdict state

- Notation: *< WI-name > : < verdict constraints > @ < input source1, input source2 >*
- Building block composition: Verdict Logic
- Characteristics: Core functionality of the What-if is the generation of advice on-demand, based on the verdict mechanism. The verdict's accuracy will benefit from making the What-If as aware as possible about the refactoring and its progress. The system can feed the What-Ifs in two separate ways; the first possibility is that the What-If asks either the user or the system the current execution point. We have worked out an example at One What-If serving more Microsteps example (subsubsection 5.3.3). Another possibility is to make the system intelligently enough to monitor the progress. For further information, navigate to the Verdict Idea (section 5.3) section

Subtype E) input only What-If detector, verdict only, the regular What-If

Subtype F) input acting as supporting child What-If, output as detector, verdict and behave like the regular type detector

- Building block composition: mixed Logic including Verdict Logic, no Selector Logic
- Characteristics: The supporting What-If may have advised on a particular case; however, the verdict did check for code states that can be necessary for further refactoring advice. The supporting child passed its findings on the output stream

5.1.7. Deriving detectors

In this section we are going to address how to derive our detectors. We suggest two sources to derive the What-Ifs from:

- Based on Conditional directives (subsubsection 5.1.7)
- Based on What-Ifs (subsubsection 5.1.7)

Following figure Figure 5.8 illustrates our approach.

Based on Conditional directives

Within section Composition of Directives (subsubsection 1.3.2), we have seen the phenomenon of the Condition Context. Remember that Fowler occasionally is referring to conditions that must be satisfied before further action is suggested. The notion of these Context Conditions appears as an explicit Mechanics step or conjunction with Actions to act as the constraints part that must be checked. These Context Conditions do not modify code likewise Microsteps, but we can see them as constraints from which we can derive helpful detectors.

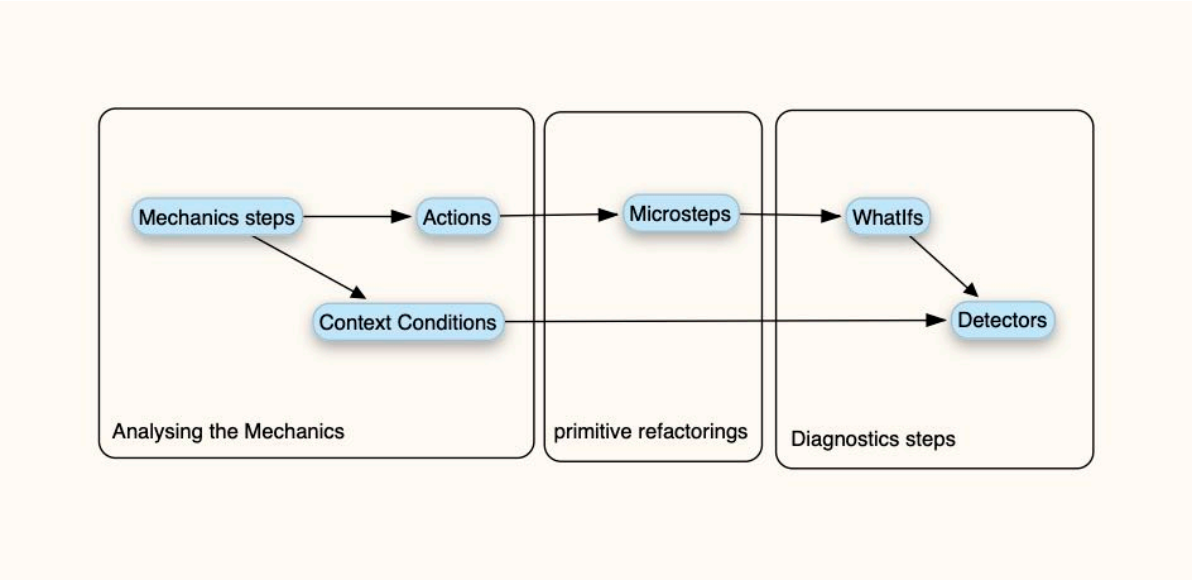


Figure 5.8: Detector development

Based on What-Ifs

Starting at Research Method (chapter 2), we have already briefly introduced the idea of Reasoning about dangers (subsection 2.3.3). Then again, we cannot do this without our detectors collecting data about the state of code on behalf of our What-Ifs. We will discuss the concept of What-Ifs (section 5.2) to the full extent, but regarding detectors, they are a foremost source for detector development.

In our refactoring discussion group, we thought about the first set of sensible applicable What-Ifs for the Rename Method refactoring during some online refactoring sessions. What-Ifs have been derived based on method-specific characteristics, like:

- visibility/accessibility/modifiability of a method
- method naming conflicts
- methods regarding overloading or overriding
- sibling methods (sharing the same superclass)
- constructor renaming
- methods within inner classes

The items from the above list are not arbitrary; namely, these are partly drafted from our Matrix in relation to Risks (subsection 4.2.3) and the (Java) Language Concepts as a problem area for the Rename Method refactoring.

The following resulting list at Rename Method What-If Use Cases (subsubsection 5.1.7) and the arguments listed above are by no means exhaustive, not only regarding potential dangers that we can encounter during a Rename action (like, for example, dynamic binding or reflection) but as well the number of refactorings covered in this thesis. The list below can be considered a mini-catalog for future work.

Rename Method What-If Use Cases

We have been collecting several matching conditions to write detectors for the Rename Method. Language concepts like overloading and overriding and core EDP[3] appliance are good sources for our investigation to provide useful What-Ifs.
Note that EDP is an acronym for Elemental Design Pattern; according to Jason McC. Smith [Smith, 2012], the EPDs are the underlying fundamental concepts of programming and software design.

Table 5.7: Scenarios #1 for What-Ifs

Matching conditions	Advice
Do we have calls inside to source class to the source method?	Inform about considering the simple Mechanics rename variant
Do we have to deal with recursion?	Inform to fix all available references for this case and continue processing
Is the source method getting called from out another method c.q. (delegated) conglomeration case?	Inform to fix all available references for this case and continue processing
In the case of abstract methods, can we speak of deputized delegation/redirection typed methods declarations?	Every abstract method declaration related implementation must be renamed as well
In the abstract class, rename its containing non-abstract member?	Precarious case to be examined further. Alter all related super and subclasses
Do we have to deal with either composition or aggregation relation?	Iterate over all method call references, change method and references
Is the target method name already an existing private method within a superclass?	Compiler problems because a private method cannot be overridden
Is target method name already an existing final superclass method?	Compiler problems because a final method cannot be overridden
Is the target method name already a static method from superclass	Compiler problems because this method cannot be overridden
Is the source method name already been declared within the inheritance tree?	Also rename the source name methods in these classes, only if the return value type equals the same or child type of parent class

From these matching conditions listed Table 5.7 and Table 5.7 we will demonstrate the case of renaming an abstract method. See Solution (section 6.3)

5.2. What-Ifs

Top of our detector evolution path is the construct of a What-If. In this section, we dive into the core of the What-If; its purpose, why it depends on detectors, and how to craft them given a Fowler refactoring.

5.2.1. What-If concept

[3]The core EPD's are: inheritance, Create Object, Retrieve (method object interactions), and Abstract Interface.

Table 5.8: Scenarios #2 for What-Ifs

Matching conditions	Advice
Which class contains the source method?	Allow rename only if signatures differ by implicit type conversion but can be conversed
Is the source method a constructor?	Rename the class first
Has the method already been overloaded?	Renaming results in program behavior preservation break
Is the target method name already know within class?	We are having a name collision
In case of a different signature, do we introduce broader accessibility?	Data hiding issues can occur
Has the source method an interface implementation?	Warn about renaming interfaces with care if possible
Do we have target method name references in the source class?	Compiler problem
Do we find a method call from inner class?	Inform to rename all cascaded source method references as well
Does the target method name already exist in any nested inner-classes?	Warn that if the target method is present in one of the inner classes (or nested inner) then the call to the source method from one of the inner-classes without explicit use of outer.inner receiver path can wrongly point to the target method

Rationale

The term 'What-If' was coined during one of the common refactoring session discussions at the Open University on premise (in Utrecht). We argued about what could go wrong during refactoring, and we started asking questions about the effects a change of code has concerning the dangers that can happen.

Case-based What-Ifs

Many different issues can occur depending on the refactoring; the change to the code, and the code context itself. For example, we can have a What-If warning about danger when we do a Rename regarding abstract classes and another one in interfaces. Here, the Microstep operation is the same for both What-Ifs, but the affected objects and conclusions are distinct.

Functionality

The functionality of the What-If is two-fold. Its core operation is evaluating possible dangers (becoming actual dangers if the refactoring is pushed through). The other main task is to report the conclusions of the evaluation. Any findings found are reported back to the user as advice, including, if elaborated, how to cope with these dangers.

Advice Templating

Advice should make sense to the user, and should contain actual and adequate infor-

mation about the dangers and possibly, their location in the source code. To achieve this, the What-If should be provided with a template advice. This template is maintained by the system, separately from the What-If. This allows for refactoring specific advice based on the template but augmented with the findings from the What-If. The refactoring processing and What-If output generation then are loosely coupled. The What-If itself is not aware of the refactoring in question until it retrieves the template data from the TAT (Template Advice Table). Separation of Concerns this way is beneficial because a small set of generic composed What-Ifs can be applied to a multitude of refactorings. We talk about the Template advice structure (subsubsection 5.2.1) of the template in one of the following sections.

Processing workflow
The What-If machinery connects the dots between the current state of selected code fragments, the identification of dangers, and the resulting advice.

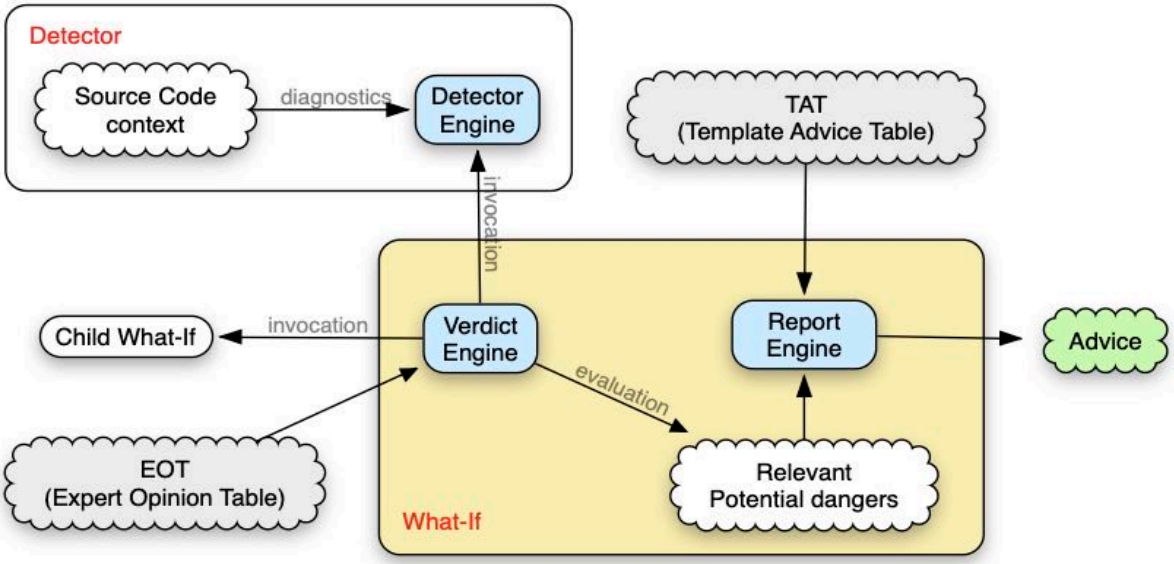


Figure 5.9: What-If processing mechanism

In Figure 5.9 we depict the processing workflow of the What-If.

- The detectors our What-If rely on delivering the code state for the Verdict engine (subsection 5.3.2) to reason possible dangers.
- The relevant dangers reflected in the outcome of the evaluation.
- That what we want to report as advice to the user.

The Report Engine fetches the template advice(s) from the TAT as discussed later, and in more detail at next section Template advice structure (subsubsection 5.2.1).

Any optional involvement of Child What-Ifs and the usage of the EOT will be discussed in Verdict engine (subsection 5.3.2). For now, it is sufficient to know that the EOT is a concept to address the problem of contradicting advice when the advice makes no sense at any moment during the refactoring.

What-Ifs and Detectors

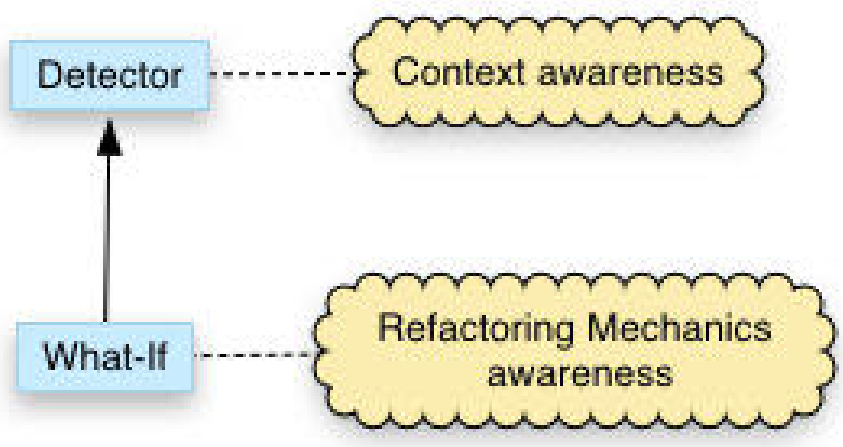


Figure 5.10: What-If detectors

Inheritance as in Figure 5.10 shows us that What-Ifs and detectors are related, because they need input from detectors for a What-If to function. Also, a What-If itself is a specialized detector capable of reasoning about the collected results from other detectors utilizing a verdict. A verdict typically serves itself with knowledge obtained from detectors (including or even from other what-Ifs). In chapter Verdict Idea (section 5.3), the necessity of a verdict mechanism will be discussed and the inner details of a detector will be explained in section Detector concept (subsection 5.1.1).

Relation to Microsteps

Regarding the process of developing What-Ifs, the intention is to create a collection of What-Ifs for all possible risks per Microstep. This means that each cell of the Microstep Matrix (being a Microstep) relates to a list of potential What-Ifs, or an empty list if case no risk is foreseen for that Microstep. Conversely, one What-Ifs can be associated with numerous Microsteps.

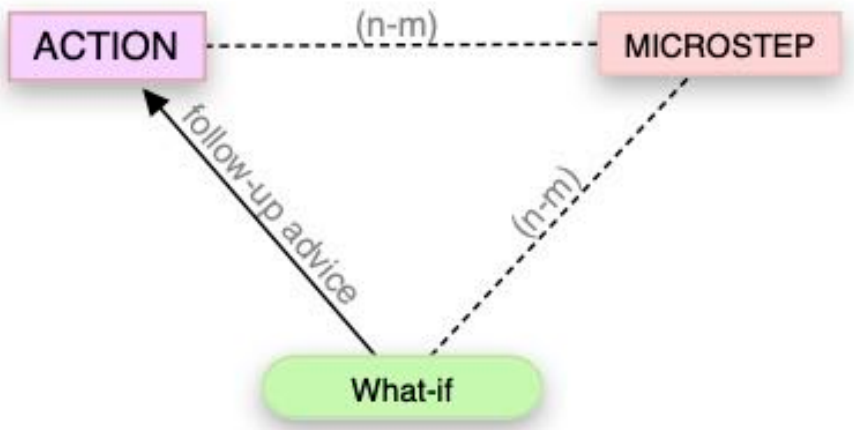


Figure 5.11: What-If loop

Besides the (*n:m*) relationship between Microsteps and What-Ifs, the advice of a What-If can contain instructions to issue other refactoring actions. These actions possibly lead to other Microsteps, creating another cycle of What-Ifs.
The Fowler Change Function Declaration (Migration Mechanics variant), for example, contains the following instruction: "If the extracted function needs additional parameters, use the simple Mechanics to add them", issuing follow-up actions.

A single Microstep leading to a series What-Ifs example
The same Add Method (AM) Microstep can be served by more than one What-If, a What-If concerning overriding methods, and another What-If concerning overloading. Previous section Rationale (subsubsection 5.2.1) portrays another example of two What-Ifs targeting different language elements. In general, the rename activity part of a method is a notable example dealing with a series of What-Ifs; in fact, we see further cases at Deriving detectors (subsection 5.1.7).

Single What-If serving more Microsteps example
Reporting about the method's accessibility and visibility can be handled by a single What-If, with only a slight difference to fill out the advice template.

Microsteps determine the applicability of the What-Ifs
Please refer to Verdict process (subsection 5.3.1) what this means for the appropriateness of the advice.

Template advice structure

Each What-If should describe (preferably) the following properties:

- What-If Id; acting as a lookup key to identify the What-If, to pass this template on to that What-If;
- Refactoring variant; the refactoring and listed variant from Fowlers refactoring catalog;
- Describing What-If matching conditions; question format style describing the applicable constraints in order to identify the dangers;
- Type; expresses the degree of the risk: informational, warning, error;
- Message entry; template-based feedback, with placeholders for actual subjects under investigation in order to report about the identified dangers and any subsequent actions;
- Tagging; is an optional argument to augment the language concepts handled by the What-If, with labels like, for example: overloading, inheritance, naming, qualifiers.

Simplified illustration of a What-If: Figure 5.12 as depicted:

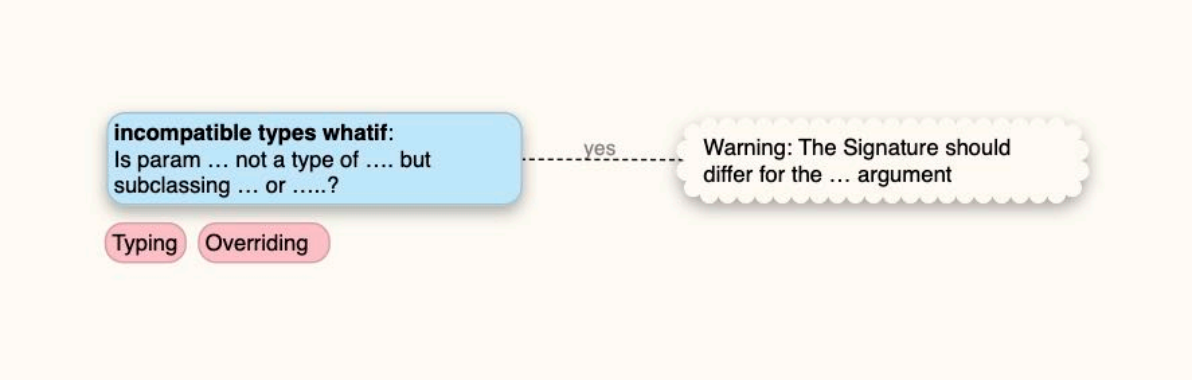


Figure 5.12: What-If example

For What-If example (Figure 5.12), we would have the following template completion in place: Table 5.9.

Sometimes, only one message entry is not enough because the advice can be different depending on the matching conditions. Take, for example, the Fowler Change Function Declaration (Add Parameter Mechanics variant). Based on the number of parameters, the message within the advice may differ completely if the number of parameters exceeds a threshold value (say 5 parameters). Fowler then advises issuing the Split Variable refactoring or the Replace Temp with Query refactoring.

The resulting template entries for above example are as follows: Table 5.10 and Table 5.11.

Table 5.9: Example What-If Template

Property	Value
What-If Id	"incompatible types what-if"
Refactoring variant	"Rename Method"
Description of What-If matching conditions	"Is param .. not a type of … but subclassing … or …?"
Type	"Warning"
Message-entry	1
Message-entry text	"The signature should differ for the … argument"
Tagging	"typing, overriding"

Table 5.10: Template entry #1

Property	Value
What-If Id	"Adding parameter count check"
Refactoring variant	"CFD-refactoring add parameter variant"
Description of What-If matching conditions	Actual #params <= 5 condition and local param check
Type	"Information
Message-entry	1
Message-entry text	"The new parameters can conflict with the local variable definition"
Tagging	"Method signature"

Table 5.11: Template entry #2

Property	Value
What-If Id	"Adding parameter count check"
Refactoring variant	"CFD-refactoring add parameter variant"
Description of What-If matching conditions	Actual #params > 5 condition
Type	"Warning"
Message-entry	2
Message-entry text	"The signature parameters must be split up"
Tagging	"Method signature"

5.2.2. What-If develop recipe

We developed a procedure for how to derive the series of What-Ifs given a Fowler refactoring.
Our procedure is based on the following assumption that; for a given refactoring and specific code context, we know in advance what the refactoring outcome of the advice should be. After the diagnostics evaluation (core functionality of our What-If), the result must lead to that expected advice.

The first two steps are relevant for embracing new refactorings from the Fowler refactoring catalog. Once we have identified our Microsteps, we proceed with the following two steps; figuring out which What-Ifs we want to support based on the risks we want to cover.

In short, our recipe to retrieve What-Ifs for a specific refactoring exists of following steps:

- Identification of the Mechanics (subsubsection 5.2.3)
- Mapping Actions to Microsteps (subsubsection 5.2.3)
- Assessment of Risks (subsubsection 5.2.3)
- Identification of What-Ifs (subsubsection 5.2.3)

The What-If develop recipe example (subsection 5.2.3) will demonstrate this procedure for a simple refactoring What-Ifs example.

5.2.3. What-If develop recipe example

Let us consider the elaboration of the "Replace Magic Literal" refactoring. This refactoring is not that difficult, but the source code seemingly remains in bad shape if we commence the refactoring without considering the possible quirks or hazards.

"Replace Magic Literal" refactoring Mechanics:

- Declare a constant and set it to the magic literal.
- Search for all appearances of the literal.
- For each, see if its use matches the meaning of the new constant. If so, replace it with the new constant and test.

Identification of the Mechanics

To illustrate the process of identifying the action kinds of Mechanics steps needed for our What-If analysis, see the example Figure 5.13 below.

Replace Magic Literal Mechanics:

- Declare a constant and set it to the magic literal.

```
Action1: Declare a constant

Action2: Assign the magic literal
```

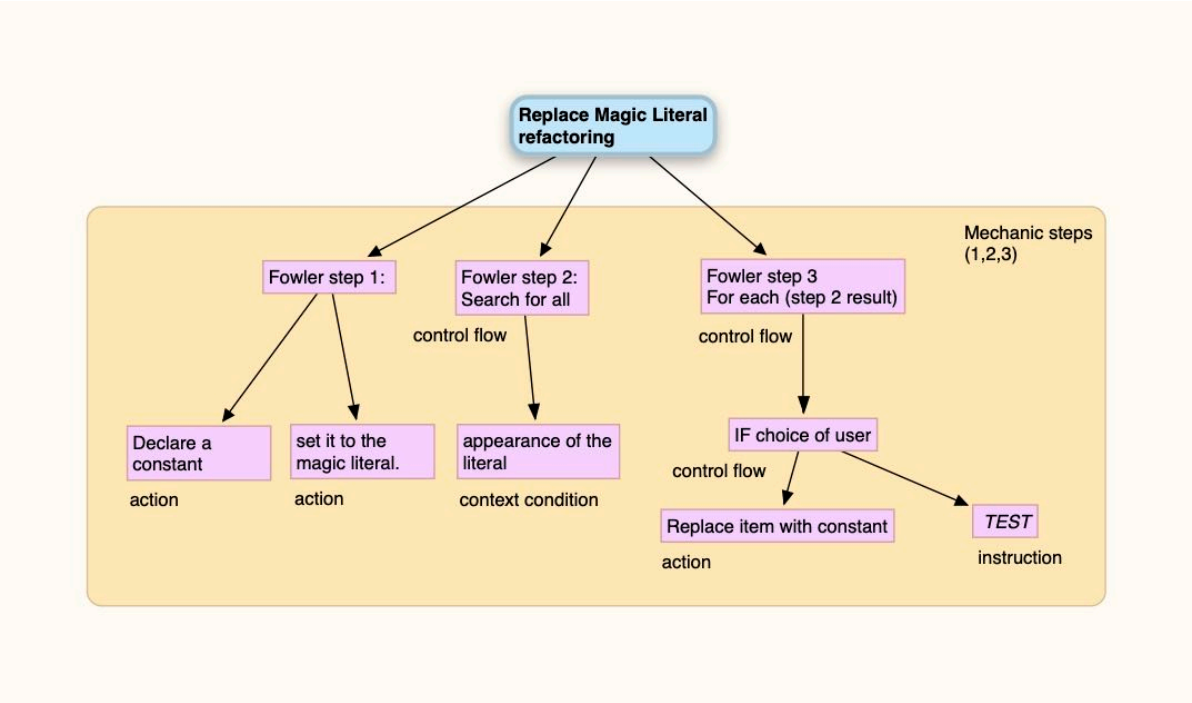


Figure 5.13: Replace literal refactoring steps

- Search for all appearances of the literal.
  Control Flow1 + context condition1:

```
Find All  similar literals
```

- For each, see if its use matches the meaning of the new constant.

```
If so, replace it with the new constant and test.

Control Flow step2: For each item in [context condition1]

Control flow step3: IF user has intention to replace item

Action3:  Replace item with declaration from Action1

Instruction: test.
```

Mapping Actions to Microsteps

We have inventoried following Actions for the Replace Magic Literal refactoring:

- Action1: Declare a constant
- Action2: Assign the magic literal
- Action3: Replace item with declaration

Next, we plot the Actions (language agnostic) onto Microsteps (language aware elements)

Depending on user intention and the location in the source code

- Action1:
  when to class: ADD field (as static const)
  when to method: ADD var (as const)
- Action2: ADD assignment statement with expression

Or the user can combine declaration and initialization as atomic action

- Action1+Action2:
  when to class: ADD field initializer
  when to method: ADD var initializer
- Action3: Change Field/Var
  target is expression: CHANGE expression statement

Table 5.12: Action to Microstep mapping

Action	Microstep
Declare a constant	AF
Assign the magic literal	AF or AF+CFI
Replace item with declaration	CMB,CFI

Assessment of Risks

The next step is to briefly assess certain circumstances that should not be neglected for the Replace Magic Literal refactoring. This quick scan could lead to the following summary of possible risks, based on either Microsteps or inherent language knowledge (or both) as possible candidates for our What-Ifs.

Extracted from the Microsteps list:

- Magic Literal and constant should have compatible type declaration. When one would replace an integer with a string constant, we have a type mismatch. Between *int* and *double* the compiler may use auto-boxing (for example, *int* vs *Int*) or implicit type casting but perhaps not to the intentions of the user.

Extracted from either refactoring literature or personal language experience:

- Overuse of constants. Fowler mentions that it is pointless to replace a literal number like 1 with the symbolic constant ONE to satisfy the possibility of doing this refactoring. Only perform meaningful replacements like PI or 'Y'/'N'. There is no real risk involved at this point, but readability might become an issue.

- Do not introduce constants that lead to naming conflicts with existing types or reserved keywords.
- Be aware not to redefine already existing constants or overshadow existing functions like *Math.PI* (which is a static member of the *Math* class). The Math class is well defined and offers superior representation of the magic literal. Why should you reinvent the wheel?
- Get away from the primitive obsession smell. Do not stick with primitives and consider using functions or objects instead. For example: *room_size >= isAverageRoom()* offers more flexibility than *room_size >= 50M3*.

Identification of What-Ifs

List of potential What-Ifs for the "Replace Magic Literal" refactoring.

Table 5.13: Replace Magic Literal, What-If 1

Property	Value
WHATIF	Incompatible types
CASE	Is the name of the to be defined constant already as an API function OR the type of the constant is not convertible (by auto boxing or implicit type conversion) compared with the literal under progress?
TYPE	Warning level
MESSAGE	Replacing might introduce incompatibility. Adjust or apply casting accordingly then.
TAGS	"Typing"

Table 5.14: Replace Magic Literal, What-If 2

Property	Value
WHATIF	Naming conflicts
CASE	Do we detect naming conflicts if we try to replace the item (from control flow step3) with the constant (identified by Action3)?
TYPE	warning or error level
MESSAGE	Replacing item (from control flow step3) with constant (from Action3) will lead to naming conflicts
TAGS	"naming, reserved words"

5.3. Verdict Idea

What-Ifs are dedicated to generating advice output. In this section, we dive into the heart of the What-If. We are going to elaborate on the idea of the Verdict. A verdict

Table 5.15: Replace Magic Literal, What-If 3

Property	Value
WHATIF	Overuse of constants
CASE	show always
TYPE	Informational level
MESSAGE	Consider only meaningful literal by constants replacements
TAGS	"General"

Table 5.16: Replace Magic Literal, What-If 4

Property	Value
WHATIF	Move away from primitive obsession smell
CASE	show always
TYPE	Informational level
MESSAGE	Fowler hints: Do not stick with primitives and consider using functions or objects instead
TAGS	"General"

is a conclusion by reasoning about the state of the code. Consider the verdict as a guardian overseer to prevent the presentation of ill-intentioned advice or improve the advice's quality outcome.

5.3.1. Verdict process

In this section, we discuss examples of why we need a verdict process.

The main drivers for the verdict process to discuss are:

- The applicability of a What-If during the whole refactoring process in general
- How to avoid contradicting advice between multiple What-Ifs, i.e., Arbitrage between What-Ifs
- False positives in case we split up a Fowler refactoring action into more Microsteps

Applicability of the What-If

During the progress of the refactoring, the list of applicable What-Ifs may change, naturally, because of the flow of actions. When for example, in a previous Mechanics step, all references to a method have been deleted, the deletion of the method itself does not inflict any more damage (concerning breaking expected functional behavior). Here, the What-If complaining about something harmless can be disregarded.

Coping with contradicting advice

Do all detected dangers impose real danger at any given time (at least during the whole process of refactoring)? No, this is not the case. Some of them are superfluous. If we consider refactoring as a series of transformations performed on the source

code, it is realistic that we temporally mess up the expecting working state of code or even the compilable state of the source code.

Under the assumption that later Microsteps will eventually eliminate potential hazards from earlier steps, we need a mechanism to control when and how the advice gets formulated. The idea of a Verdict engine (subsection 5.3.2) implements this mechanism.

Take for example the (Extract Function) Mechanics step, adding a new function. Here, adding a non-void method (via AM Microstep) with empty body introduces a flaw. However, this (temporal) situation should be neglected because, as part of the refactoring course of action, this hazardous situation gets fixed as soon as we add the proper return statement (invoking CMB Microstep). The AM Microstep is part of a conducted case in Verdict example (subsubsection 5.3.3).
Another example is if we want to give a method a new body that might flag for danger to some detectors because of the existence of the old body (for instance, during selective copying of statements to the new body). Nonetheless, this is a temporal danger we can safely ignore if resolved eventually.

Arbitrage between What-Ifs

False Positives

Regarding false positives, conflicting information might likely occur when you check for the same conditions before and after a Mechanics step. In the case of renaming a variable, for example, the new name may not yet exist since the old one still exists. After the refactoring, you expect the new name to exist (and the old one no longer, of course). The final warning that the new one exists no longer applies.

Arbitrage is needed if we do actions like moving code around or changing objects by removing it first and later adding the new object. This temporary removal is necessary 'evil' to obtain the purpose of the refactoring. However any advice here can be safely ignored.

To prevent those false positives emitted from the remove related Microstep, we put an overseer What-If at work; the Arbiter What-if (subtype E detector, see Detector sub-typing (subsubsection 5.1.6)). The task for the arbiter, as shown in Figure 5.14, is to overrule these false positives from child What-Ifs (subtype F, see also Layering model (subsubsection 5.1.6)).

The idea is that the Arbiter What-If can decide either to:

- simply not invoking the false-positive emitting What-Ifs,
- or to pass on directives to (be obeyed by) invoked children,
- alternatively by augmenting the advice from the child What-Its, adding extra advice.

With the aid of an Expert Opinion Table (EOT: Rationale (subsubsection 5.2.1)), we can feed the arbiter how verdict: Table 5.17.

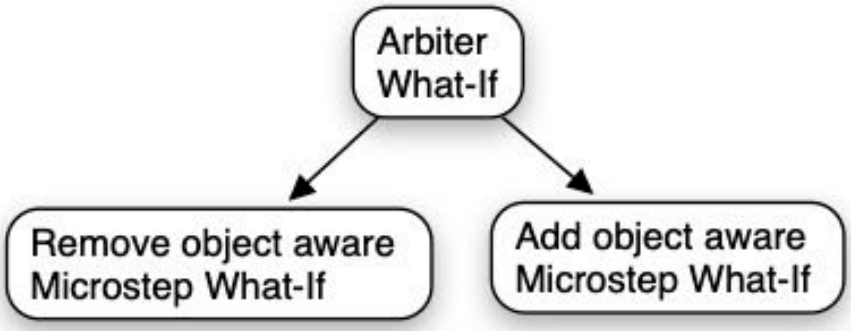


Figure 5.14: Arbiter What-If

Table 5.17: Arbitrage decision table example

Microstep to be executed	What–If to be invoked	Verdict level [1–5] 1=ignore	Arbiter What–If
Remove operation	Remove object	5	Usual invocation
Remove operation	Add object	1	Object already exists warnings should be suppressed
Add operation	Remove object	1	Object already removed, ignore the Remove object What-If
Add operation	Add object	5	Usual invocation

5.3.2. Verdict engine

The What-If is functionally extending the Detector; see inheritance model in Figure 5.10. Hence, the What-If inherits the property of code context-awareness. Furthermore, we can distinguish the What-If by verdict level. Each increasing level broadens the awareness capabilities, and each verdict level of operation gains the features from the previous level, as seen here at Figure 5.15.

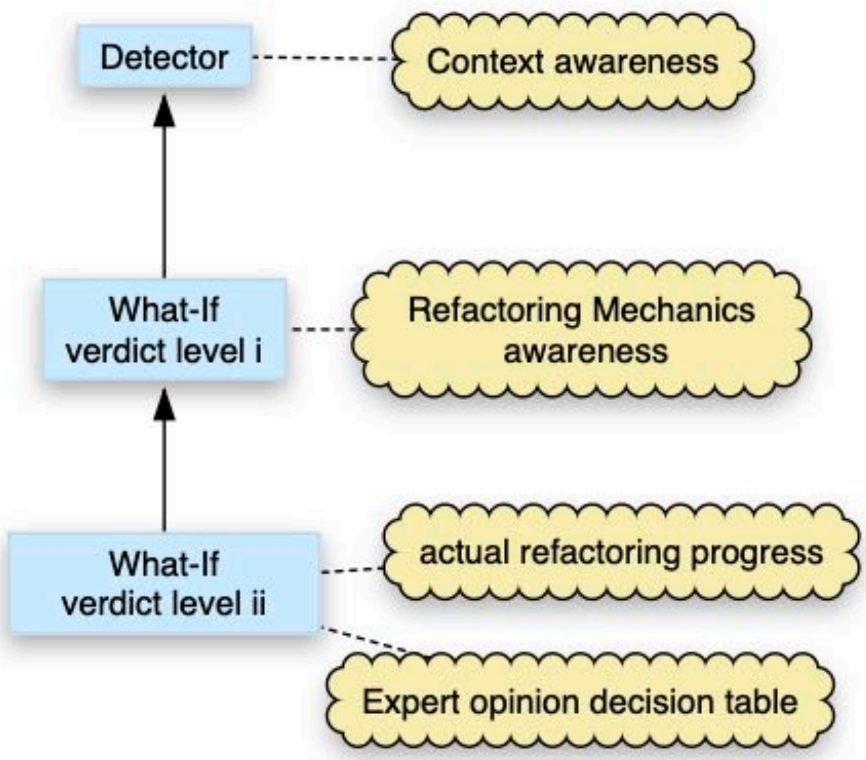


Figure 5.15: Verdict level of operation

What-If verdict level ii corresponds with What-If subtype E, Detector sub-typing (subsubsection 5.1.6).

Reasoning mechanism

In order to reason about the output of detectors, the verdict engine needs to be aware of the following conditions:

- current code under investigation by detectors
- the What-if is aware of which dangers for which refactoring case

- Current Mechanics step and Microstep execution in progress

The verdict engine needs to be aware of how to treat the Mechanics Steps in regard to the code context. For this to function, detectors deliver information about the state of the code extracted from the current code context.
The What-If (where the verdict resides) knows which refactoring dangers it covers and, therefore, can decide how to interpret the input from the detectors. However, this interpretation is sensitive to false positives because it must know the circumstances for which the advice is meaningful.

Because the verdict engine knows 'when' (bullet 3) and 'which' (bullet 2), Microsteps can potentially lead to dangerous situations; based on the analyzed (bullet 1) Mechanics steps action directives, the verdict engine can decide accordingly, generating tailored advice.

We distinguish three levels of operation for the verdict engine. Regarding this study, we have full support for level I. For level II, we worked out the expert opinion proposal next section Weight Level arbitrating (subsubsection 5.3.2). If the demand includes dynamic determination capabilities, we arrive at level III. But this level of sophistication requires the implementation of RQ4: Monitoring integration (Future Work) (subsection 2.1.5).

Verdict reasoning levels:

1. In its simplest form, executing a verdict is no different from logically combining (one or more) dependent inputs from detectors to be called. The decision of advice is the What-If's responsibility (and main task) and should therefore not be left to an 'ordinary' combining logic detector.

   What-If subtype F operates on this level Detector sub-typing (subsubsection 5.1.6).

2. In a more complex situation, the What-If must be aware of the refactoring and the refactoring stage. We can achieve this by having the system determine the progress per Mechanics step, whether or not with the help of the student which Mechanics step from the list of steps per Refactoring the student will be working on. Knowing the stage is important because advice might no longer apply. What-Ifs could also contradict each other based on the user's actions in combination with the progress.

   The arbiter What-If Arbitrage between What-Ifs (subsubsection 5.3.1) (subtype E) operates on level ii or lower-level i Detector sub-typing (subsubsection 5.1.6).

3. Suppose we want to be able to overview the refactoring process and to measure progress dynamically. In that case, we need to track all actions taken. This extra information has to be shared between every What-Ifs with the aid of an additional input stream. This steam needs to be maintained by a kind of supervisor What-If. Such a sophisticated What-If is needed for continuous monitoring of the refactoring process. However, monitoring the progress is a future enhancement, out of scope for this thesis.

Weight Level arbitrating

A suggested solution to address the issue of contradicting advice is to maintain an Expert Opinion Table (EOT). The verdict then is based on the resulting weight information from this table. It contains configurable information, supplied and maintained by a refactoring expert (perhaps the designer or implementor of the What-Ifs).

This EOT table hints to verdict engine the expectation of how practical (or wishful) the What-If's advice will be. As a rule of thumb, expectations are based on the assumption that any What-If, not associated by the triggering Microsteps at a specific Mechanics step, is under suspicion.

What do we need to collect for the refactoring to build up the verdict level decision table?

- The list of Mechanics steps in execution order
- Which Microstep invocations from which Mechanics steps
- For the What-Ifs, the relation to the Microsteps they cover
- The runtime registration of the current Mechanics step to be executed by the user.

Verdict weight levels in relationship to the response of advice for the Arbiter What-If and suggested reactions:

1. regular advice is inappropriate, will be treated as false positive then
2. ignore/suppress child advice, or

   if the threshold value is in use by the Arbiter What-If: only accept advice if all the threshold levels of underlaying What-Ifs score higher or same value,
3. ignore/suppress child advice, or

   if the threshold value is in use by the Arbiter What-IF: only accept advice if all the threshold levels of underlaying What-Ifs score higher, or

   the child advice should only be given by a superseding What-If (pass on to supervisor for decision), typically do not invoke the child What-If
4. the What-If can decide for itself if it suits advice or pass it on to the caller What-If
5. advice may be given on any occasion (including hints or best-practice type of advice)

See Arbitrage between What-Ifs (subsubsection 5.3.1) for a coarse-grained approach in which we only use the values 1 and 5.

At One What-If serving more Microsteps example (subsubsection 5.3.3), we show

an example of threshold values. The threshold level is a value between 1 and 5 maintained by the Arbiter What-If itself. If any of the child What-Ifs does not meet the threshold level, the Arbiter What-If can overrule the child What-If according to the reactions described at Arbitrage between What-Ifs (subsubsection 5.3.1).

5.3.3. Verdict examples

Verdict example

We use an oversimplified example drafted from the Change Function Declaration Refactoring migration variant to demonstrate the Verdict process (subsection 5.3.1) phase.

Initial code:

```
class Ex {
   public int test() {
      System.out.print(" test()\n");
      return 42;
   }
}
```

Finalized refactoring:

```
public class Ex {
   public int newtest() {
      System.out.print(" newtest()\n");
      return 42;
   }
}
```

The illustration at Figure 5.16 shows the Identification of What-Ifs (subsubsection 5.2.3) result of a fictional 'demo intentional' refactoring, with the following meeting conditions per What-If:

- Name Class What-If; checks on name clashes for the new method name,
- Safe removal What-If; checks if the old method eventually gets removed,
- Valid return What-If; checking for the proper return value.

AM Microstep execution:

The Add method Microstep creates a new method with the same signature and visibility as the original method. The new method gets an empty body (or with a dummy return value statement at most (if we want to avoid compile errors right away and if we decide that the AM Microstep should be equipped with functionality to cope with all kinds of return types). Note that Fowler ignores all these details and talks about adding a new function.

The order of execution in this illustration is the order of appearance from left till right. Let us assume for the sake of this example that the refactoring practitioner is busy with the CMB Microstep transformation (note that the Valid return What-If takes the verdict outcome of the Name Class What-If into consideration because the new method must be in place).

Consider the Remove Method (RM) Microstep. Removal seems appropriate only when

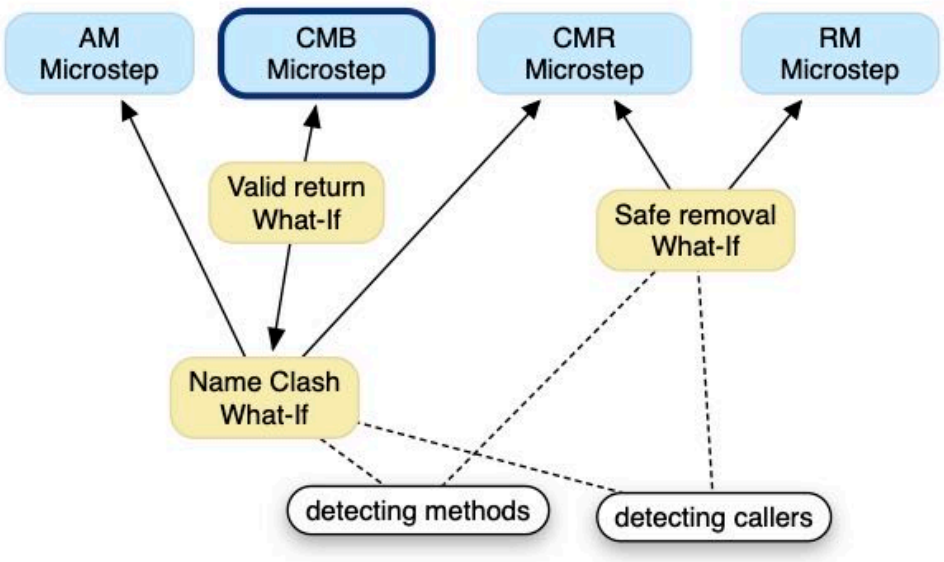


Figure 5.16: What-If scope of execution

there are no method calls left referencing the to-be-removed method. Our Safe removal What-If advice makes only sense when the refactoring practitioner finishes the action invoking the CMB (Change Method Body) Microstep.

The verdict should be informed about the progress of the execution of the refactoring. The verdict of the Name Clash What-If makes sense just before the AM execution to warn about the potential dangers in case of a naming conflict.

One What-If serving more Microsteps example

For example, imagine that the student is busy executing a fictitious refactoring variant comprising Steps 1, 2, 3 in consecutive order.

We present a fictional refactoring at Figure 5.17 with the following Microsteps participation:

- Microstep CM (Change Method) invocation from: Mechanics step 1, 3
- Microstep CMM (Change Method Modifiers) invocation from: Mechanics step 2
- Microstep CMB (Change Method Body) invocation from: Mechanics step 3

What-If relationship towards Microsteps:

- What-If 'Method consistency' advises about microsteps X and Y

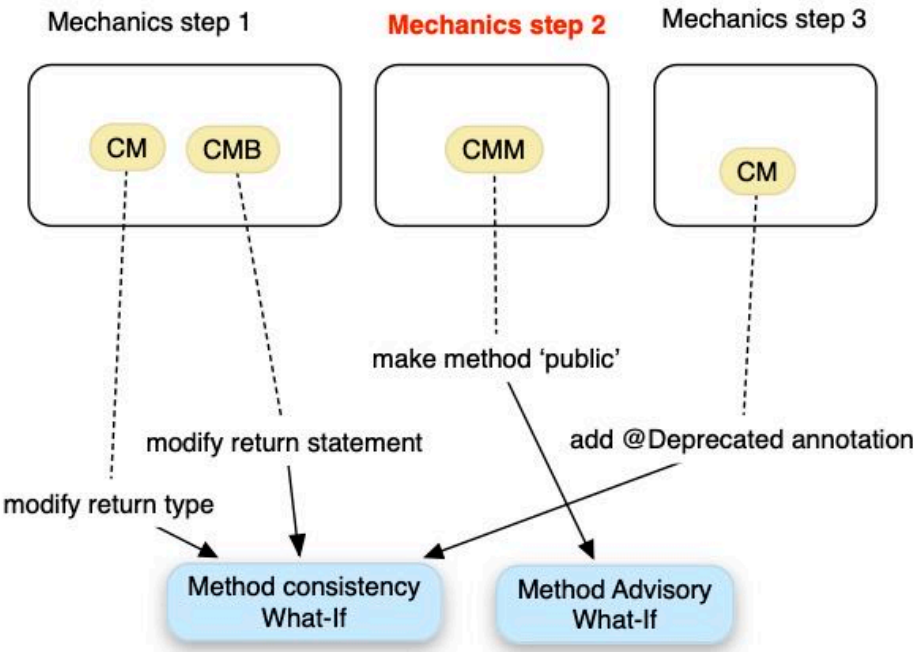


Figure 5.17: What-If serving more Microsteps

- What-If 'Method Advisory' advises about microstep Z

The pointer to current Mechanics step execution, the step that is going to be executed by the refactoring participant is: Mechanics Step 2.

Our refactoring expert decided to maintain a table with graduated values between 1 till 5 (nd anything in between (1= ignore the advice, 5 means the opposite). So the code may decide the verdict based on the level of importance from the invoking Microsteps.

Crafted expert based verdicts:

Table 5.18: Expert Opinion Table example

Mechanics step#	Microsteps	What–Ifs	Verdict level [1–5] 1=ignore
1	X	A	5
2	Y	A	5
3	X	A	5
3	Z	B	5
3	Y	A	4
2	Z	B	1
2	X	A	2
1	Z	B	1

Code excerpt for What-If 'Method consistency', how the Verdict proceeds based on the expert opinion hints:

```
1    Current  = getcurrentMechStep();  // gives 2
2    VerdictLevelSet = MS4WhatIf(me); // gives the set [2,5]
3    IF VerdictLevelSet.VerdictLowestlevel > 4
4        DoVerdict();
5     ELSE  IF  VerdictLevelSet.VerdictLowestLevel <3
6           AND VerdictLevelSet.VerdictHighestLevel < 5 THEN
7         ........ etc. etc.
```

6

Prototyping the Framework

6.1. Envisioned Architecture

6.1.1. Requesting Features

We have set a number of function-specific requirements for the implementation of our prototype tool:

- Development of our Detectors and What-Ifs should be fast and easy.

  To substantiate rapid development, we have chosen the Prolog declarative language for the capability to express our code in Prolog facts and rules, as does our tool-specific AST. The rationale why we want to have a special-purpose AST has been explained in Conceptualizing the AST (section 6.2);

- For learning effects and detector development, a facility to toggle off or on the execution of any detector;

- As a result of the analysis, the tool should publish the advice per What-If, in textual form, including the affected source code lines to which the advice relates or refers.

6.1.2. Guidance staging Model

With the above feature-specific demands in mind, we defined a Guidance staging model that adheres to our intended refactoring flow, divided into several phases.

Refactor phase

Refactoring is an integral and repetitive part of the system. Execution of the refactoring steps enables getting feedback. Initially, our scope for this study is only to support feedforward advice. Feedforward can be obtained before refactoring but after the analysis phase.

Parse phase

Parsing code is an obligatory and repetitive phase. Once the student saves the selected source code project (a prepared tutorial lesson or self-made code), the IDE

reacts (default behavior) by rebuilding to byte code. Hooked onto this build event, the tooling will have the chance to generate our tool-supported AST representation, used by the analysis phase.

Analysis phase
The order of detector execution is based on its position in the calling chain. Logic is programmed in the tool's native language (Prolog). In the case of Prolog all code albeit detector/Verdict engine (subsection 5.3.2) logic, assisting code like libraries, etc., must be 'consulted' beforehand.

Report phase
It would be nice if we offer the student the possibility to selectively influence to the outcome of a report, that is; which of the What-Ifs will contribute to its list of advice in the resulting report. Our tooling solution controls this by setting a checkbox per contributing entry in the GUI.

Setup phase
Part of the setup is constructing all detector logic, the What-If verdict logic, advice templates, utility functions, and perhaps some configurable data. Ideally, the tooling directly provides all the functionality. How we can deploy and configure the tooling during setup was very determinative for selecting the tooling.

6.1.3. Tooling platform

To exercise the Fowler refactorings, we need to specify a defined software platform that supports the language to practice. For this study, we target the Java language.

Our proposed tooling platform comprises different layers of integration with the IDE. Layering make it easier to replace some constituents of our tooling solution with other alternatives, like, supporting other IDEs or mechanisms for other AST representations.

The overview in Figure 6.1 shows information about the deployment of the prototype tooling but can very well be used as a basis for another alternative implementations. For suggestions, see also Future Work (chapter 9). The foundation of our architecture depends on the availability of the target language and a tool-specific host language. At this time of writing, the setup is based on actual available versions of the Eclipse IDE edition and current Java SDK and SWI-Prolog install-base.

Out of the box, any Java source code project (yellow colored) is built and parsed (after a save action) into Eclipse's own AST structure. However, if you want to query this AST, you need to have extensive knowledge about the JDT toolkit. Luckily we can circumvent this by installing our own dedicated plugins to enable the wanted features. The (blue colored) object represents the creation and deployment of our detector/What-If chains and individual interpretation thereof. Perhaps shortly, we should use separate optionally crafted tooling for rapid prototyping the detectors, but we are now developing them as Prolog modules carefully by hand.

The aforementioned Prolog modules are the core of our tool, depicted by the (red-

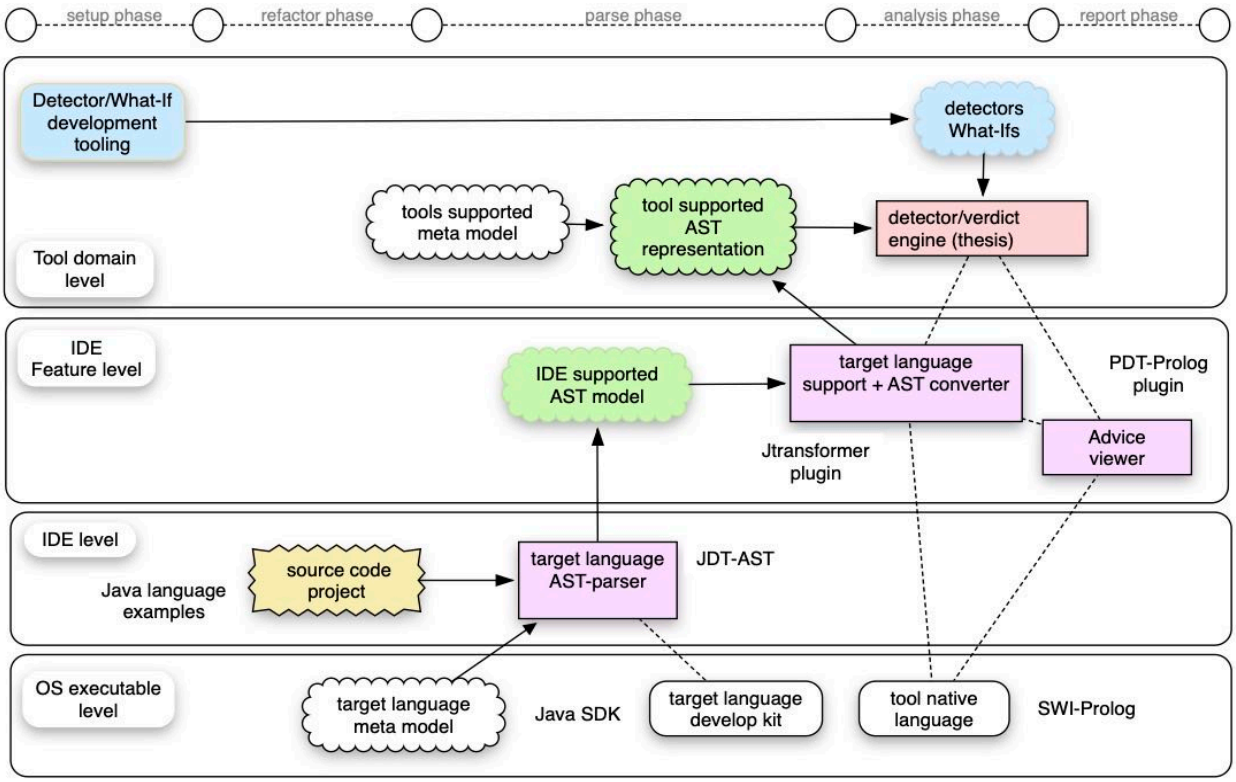


Figure 6.1: Used prototype architecture

colored) Verdict engine (subsection 5.3.2). Here we construct the logic of every detector and What-If into Prolog clauses. This part is also a candidate for future improvement.

6.1.4. Tooling solution

Our tooling of choice should incorporate a complete development stack for actual editing, run-time building, and testing of Java code. Nowadays, any popular IDE will suffice, as long as they are extensible. For our purposes, we have chosen[1] the Eclipse IDE because of its freely available add-ons. On the IDE level, the Eclipse IDE serves our needs to act as a host for both Java source code programming and the possibility to develop the ingredients to enable the guidance of students.

Requested IDE feature level solutions:

- JTransformer, our principal Java code to Prolog facts and rules plugin, on its turn it relies on the internal Java AST representation. Both the ASTs listed (depicted as green clouds) are generators based on the actual code context
- PDT Prolog plugin enables for bridging between the SWI Prolog language engine and Eclipse IDE host

JTransformer's key aspects:

- Translates JDT specific AST into own proprietary AST node format
- Proprietary AST inspector for actual Java code inspections
- Analysis and Result views (used to present the What-If advice and results found by the detector)

PDT Prolog Eclipse perspective

- Native SWI Prolog Language integration
- Console, for REPL mode execution of Prolog, including debug/trace support

6.2. Conceptualizing the AST

We opt to maintain an AST separate from the one internally used by the selected IDE. This way, we acquire optimal control over structure and content independently of the IDE's AST implementation. Another advantage is that we can abstract away from the complexity most of the AST implementations have. A simpler, more adequate model makes it easier to customize and develop utility libraries (like navigating the structure). A tool-specific AST powered by a Property Graph Database, for example, might perhaps be a future candidate tooling solution because of this decision. However, a drawback of maintaining an additional AST is the consumption of time it takes to transfer data to our own AST.

[1] Other IDEs like Jetbrains' IntelliJ or MPS would have taken us more time to prototype in.

6.2.1. AST Modeling

We adopted the AST design from the accompanying documentation of the JTransformer plugin (introduced at Tooling solution (subsection 6.1.4)). This design maintains a structure independent of the target language. However, the concept is not unique since it resembles the standard RDF triplet notation for objects from W3C.

Meta Model

In Guidance staging Model (subsection 6.1.2) we see that the IDE's project source code gets parsed into the AST. The source code is displayed as nodes (and relations between them) in the AST model. The target language syntax, however, is defined by the AST Meta Model.

Regardless of language syntax, for example, class or method, the Meta Model conforms to a uniform description as depicted by Figure 6.2.

Nodes

The tree is built up from nodes and edges. Every node has a primary key called 'id'. This id uniquely identifies the entity object. Two methods would therefore have different id-values in the AST model to make them distinguishable. For instance: *methodT(#1,.'n'...)* en *methodT(#2,.'n'...)* refer to different entities, although of the same kind (method objects) and with the same method name.

Edges

Most of the nodes have outgoing 'parent' edges except for root nodes. Packages, for example, are root nodes. Other node types like classes, methods or statements may express other non parental relationships. These nodes have other than parent edges[2].

Property Objects

Besides the obligatory property object: 'ID', nodes may contain additional property objects of either type, attribute or type relation.

- The attribute type of objects property does not refer to other nodes.
- The relation type of objects property, however, does refer to other nodes. The name of the property also becomes the edge name. In order to build a tree of nodes, the property named 'parent' points to its parent node.
- Nodes without property objects other than ID behave like flags. For example, the interface node type marks the class as an interface.

Meta Model-level example

If you remember the Method and Fields syntax (subsubsection 4.3.2), for example, all the method aspects, such as return type, name, method signature, the AST Meta Model does have counterparts.

[2]Besides 'parent', the AST also uses 'ancestor' edges for internal usage.

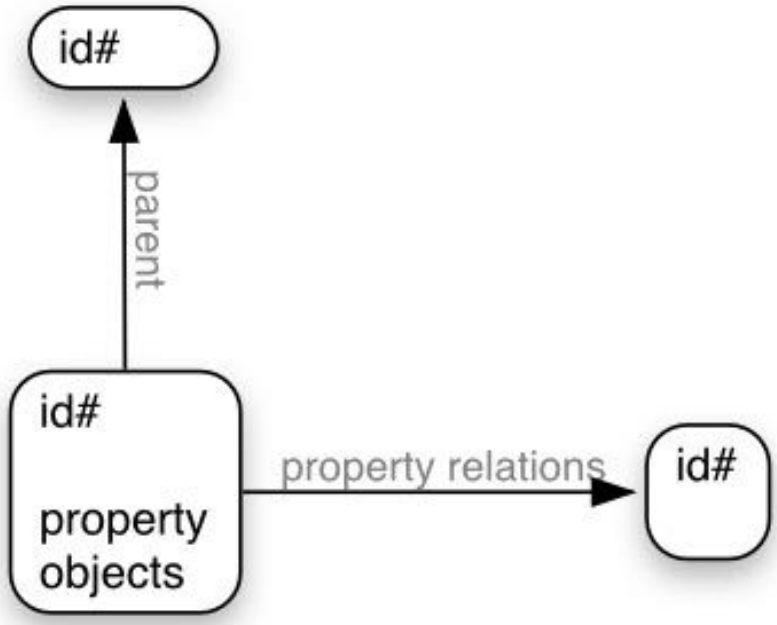


Figure 6.2: AST meta model

The 'methodT' node-type, for instance, contains the following information to define the method syntax:

1. the unique ID of this method
2. the ID of the class containing this method
3. the name of the declared method
4. the list of IDs of the method parameters
5. the id of the return-type of the method
6. list of IDs of checked exceptions thrown by this method
7. the list of IDs of the type parameters
8. ID of the block containing the method body

The method syntax Method and Fields syntax (subsubsection 4.3.2), visually reflected as AST Figure 6.3, comprises the method node type composition to make up the method definition. You may have noticed that all the blue-colored nodes have inbound edges because the method node type (pink colored *MethodT* node) has containing property objects pointing to them, except for the modifier type of node with an outbound relationship with method. Modifiers are not restricted to methods only and, of course[3], more than one modifier (combinations of access /non-access) is allowed.

Method in use example

A practical demonstration Figure 6.4 of the method node type is when we perform the Rename Method refactoring. The CM Microstep changes the name of the method, acting directly on the *methodT* node type. To assure that all invocations to the method still apply, the CMR changes all method invocations to the new method name. As we may observe in Figure 6.4, the CMR Microstep operates at the level of the *CallT* node type. Notice that the method call always makes use of a receiver object specified by the *identT* node type. Method invocation is described in Method Call syntax (subsubsection 4.3.2).

Java Language Concepts

When we refactor, we have to deal with the concepts of the language. The overview Figure 6.5 maps the concepts as listed next to the relevant AST node types equivalents and cooperation between those nodes

- Scope (class, method, body, statements); *examples: 1a,1b, 2b, 3a,3b, 4, 5*
- class Containment, Inner/Outer Classes; *example: 1b*
- Referencing, method calls (receiver); *example: 5*

[3]Modifiers provide (non) access functionality for classes, interfaces, constants, fields, methods, parameters and import statements.

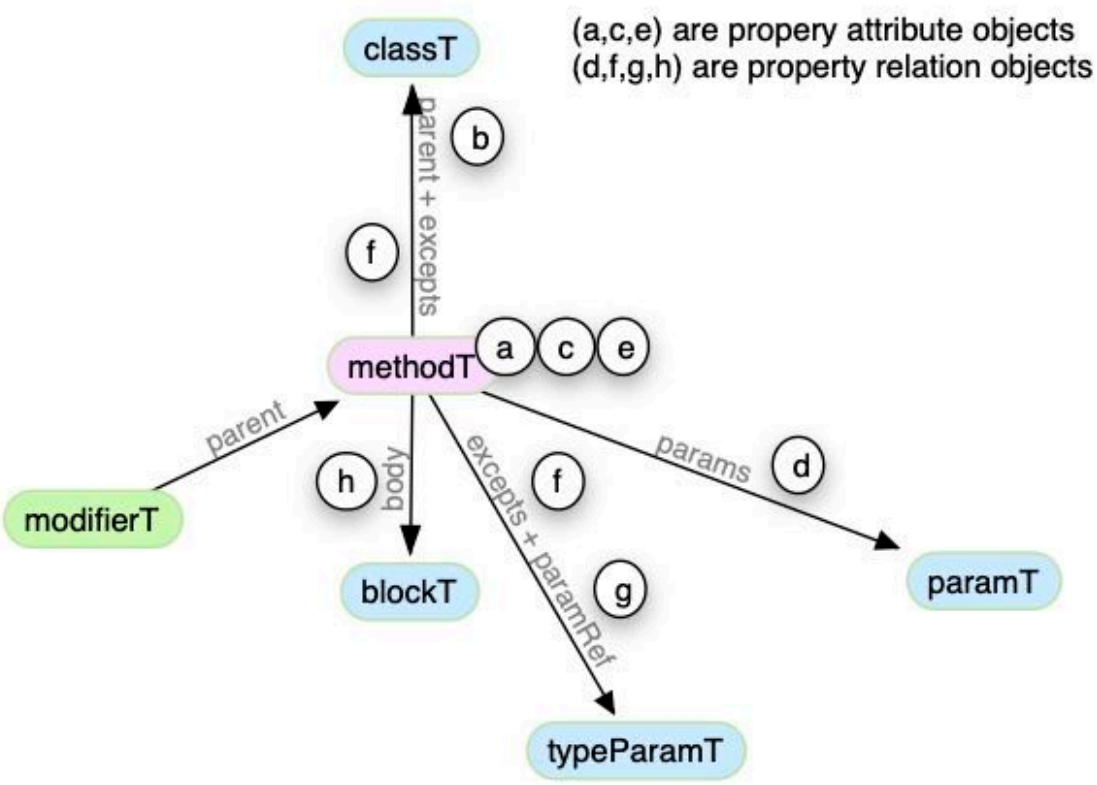


Figure 6.3: Method syntax node type composition

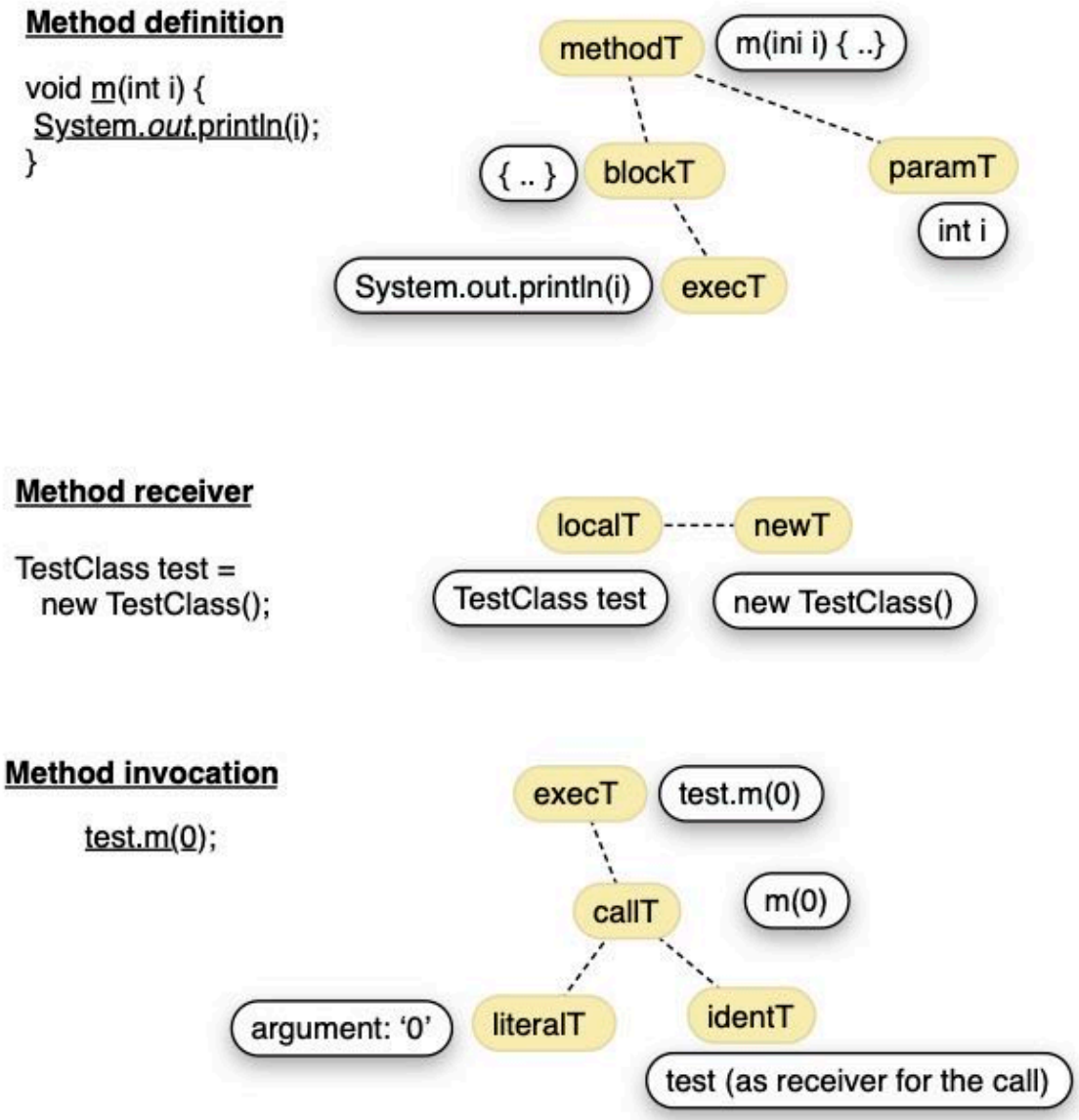


Figure 6.4: Method invocation AST excerpts

- Inheritance of Classes, extends mechanism; *example: 3a*
- Implementation of Interfaces, implements; mechanism: *example: 3b*
- Typing, specifying; *examples: 1c, 2a*

For instance, finding a method within a particular class, is an example Constraints logic (subsubsection 6.2.2) for 1a.
Class inheritance case 3a is shown at Navigation logic (subsubsection 6.2.2). Even more cases will be demonstrated at Prototype demo

6.2.2. AST representation

Constraints logic

In the Figure 6.6 example, we have defined a mini detector that wants to find all the methods 'n' matching class node. The black-box notation is something like: method.n?@class[4]. The code context consists of class with two methods 'n' and 'm'. To oversimplify this AST representation, only the three nodes (for class and methods) are contained. The AST query, in this case, has to match all those nodes of type method, having 'class' as the parent node and the method-name equals 'n'. The answer will be put into a list with the id of the matched notes. In our case, only one node will match as expected.

The Figure 6.6 example is based on how in Prolog you query the AST fact-base. But one can imagine supporting library functions to ease the process of querying the AST.

Navigation logic

To do AST navigation, we traverse the tree by using the 'parent' edges. Independently of the target language, in an AST, all nodes except for the root node have ancestor nodes, and the root node all up to the leaf nodes have descendants. Because the tree can have multiple node levels, it may be necessary to visit numerous connected nodes. For example, suppose we want to find out the top-source level class in the case of the abstract classes diagram Complex refactoring (subsection 1.2.3) renaming a method of a class at a certain level. In that case, we need to traverse the parent edges according to transitive closure to reach the level class.

In the preceding section Java Language Concepts (subsubsection 6.2.1) we saw that the *extendsT* node type enables the Java inheritance concept for classes and *implementsT* node type for interfaces. In both cases, the 'parent' edge points to the subclass or implementing class, and the 'super' edge to the superclass or interface.

We demonstrate the inheritance mechanism by a white-box detector example, with supplemental Prolog code (red text) for the detectors. See Figure 6.7.

The elegance of the implementation[5] choice of language is that it is very short but descriptive, similar to our detector notation language Blackbox notation (subsec-

[4] method–n@class will do as well; notation is not of importance as long as its purpose is obvious
[5] The tool–supported language Prolog as a declarative language is of great aid in describing what you want.

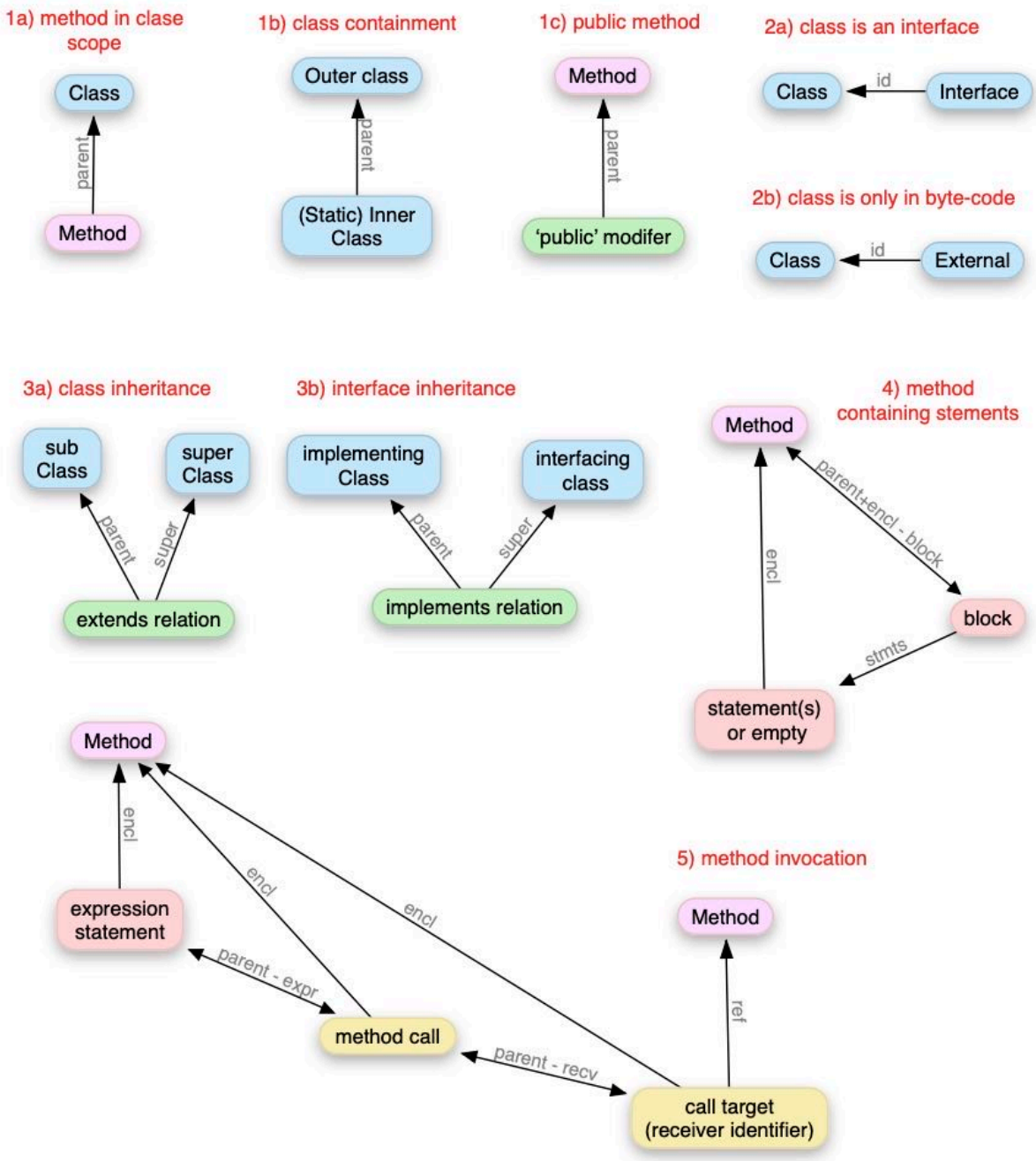


Figure 6.5: Java concepts AST representation

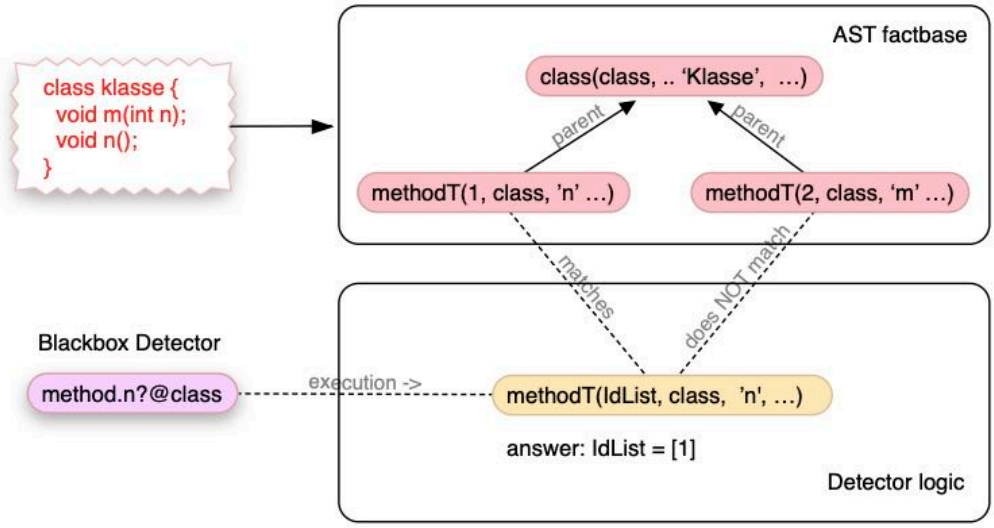


Figure 6.6: White-box detector implementation

tion 5.1.4). The What-If task here is to issue warnings when the source method is defined in an interface because the method definition in the interface should also be renamed.
The What-If answers by combining the result from two detectors; the target method enclosing class (case 1a) in the inheritance chain (cases 3) together with its declaration in the implementing interface class (case 2a of Java Language Concepts (subsubsection 6.2.1)).

Let us examine the implementation in more detail, based on the given AST for the following code. The demo solution here is simplified because it is a fit-for-one-purpose-only solution. We simplified because we should check for equality on method-name and method signature in case of overloading. A more appropriate design taking the overloading and overriding principles into account is presented at EF-refactoring demo 1 (subsection 6.3.1).

```
1 public interface TestClassInterface {
2   void m(int m);
3 }
4
5 public class TestClass implements TestClassInterface {
6   @Override
7   public void m(int m) {
8     ;  //Some very important code ;-)
9   }
```

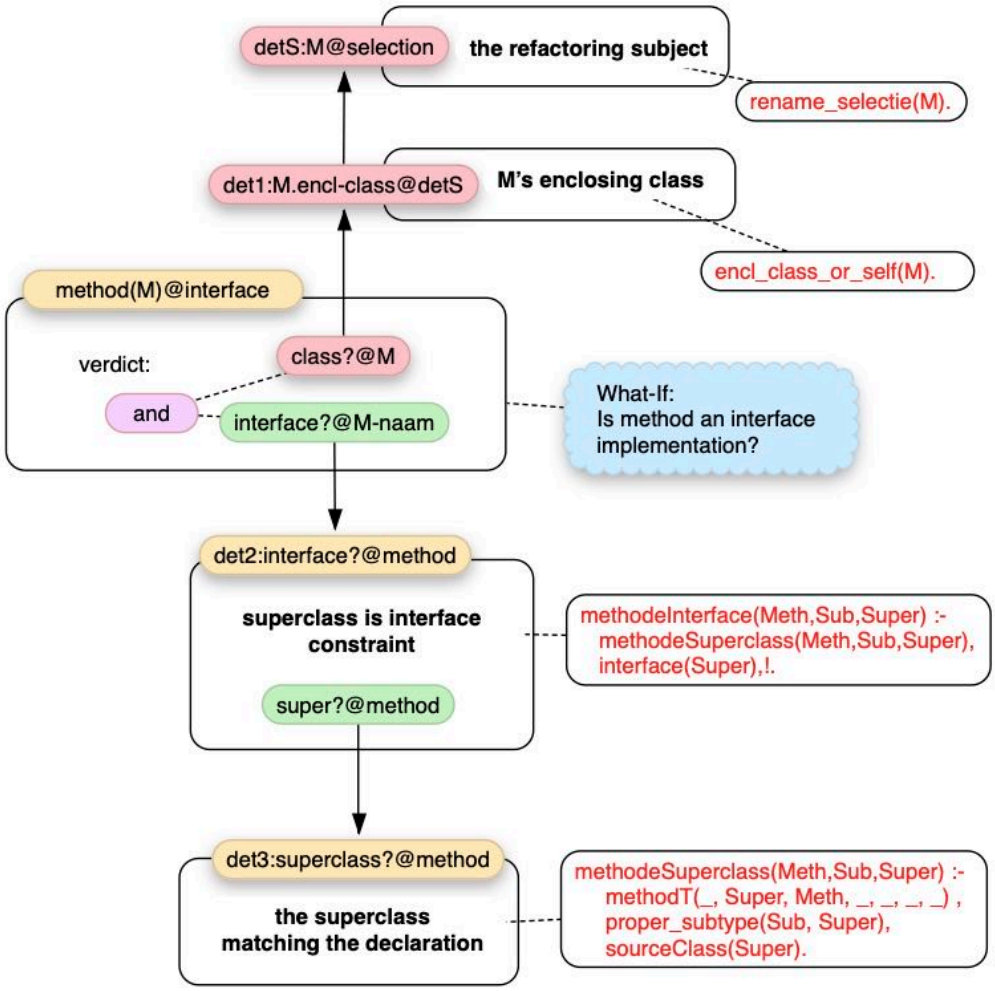


Figure 6.7: White-box What-If example implementation

```
10 }
```

Listing 6.1: Override method listing

The solution will be illustrated below Figure 6.8 in white-box format. The clauses that are composing the Prolog detector's predicates are explained in Table 6.1.

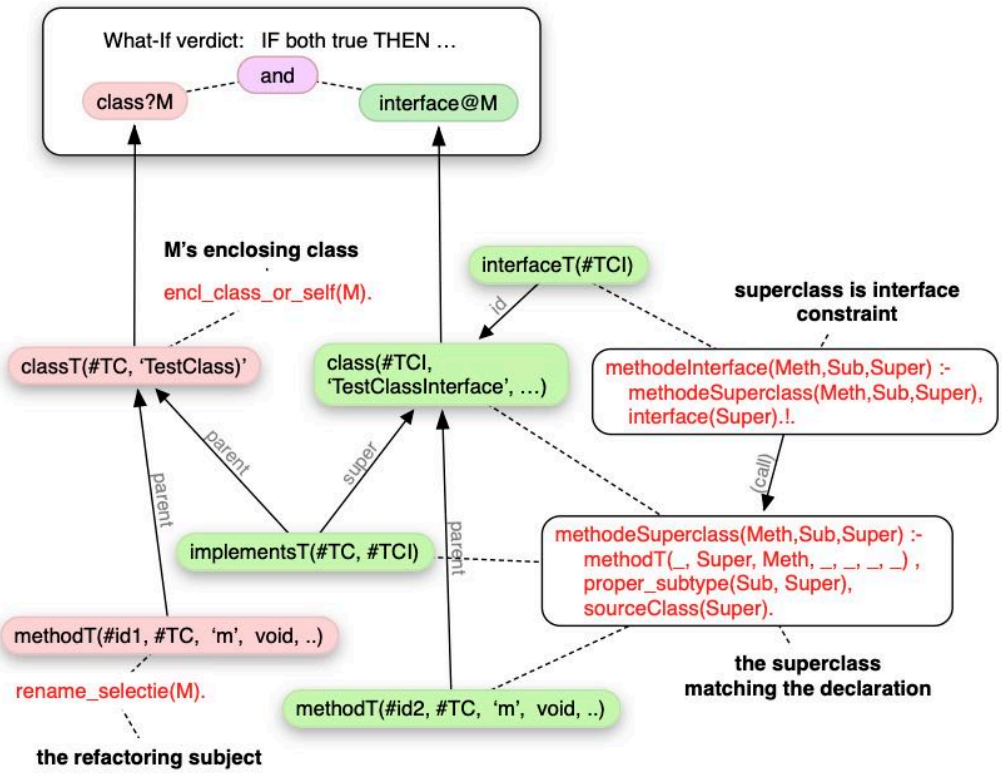


Figure 6.8: Detectors matching example implementation

Prolog clauses for our example detector logic:

6.3. Solution

In this chapter we demonstrate the construction of detectors to support renaming a method.

6.3.1. EF-refactoring demo 1

The Extract Function (EF) refactoring is part of the umbrella Change Function Declaration refactoring (CFD-Migration Mechanics variant) because CFD is the principal refactoring for renaming a method.

For Figure 6.9, we want to rename the method *m()* in the *Sub* class. Because this method overrides the method from the *Sup* class. Let's focus only on renaming the

Table 6.1: Method override internal detector logic

Clause	Explanation
rename_selectie(M)	Matches the method node #id1
encl_class_or_self(M)	Matches the enclosing class in which #id1 resides
methodT(..Super,Meth,..)	Matches all superclasses having the method name equal to Meth given that Meth gets passed the value of 'm'
proper_subtype(Sub,Super)	Sub must descent from Super
sourceClass(Super)	We are not interested in byte-code only classes like Object
interface(Super)	The superclass must be an interface

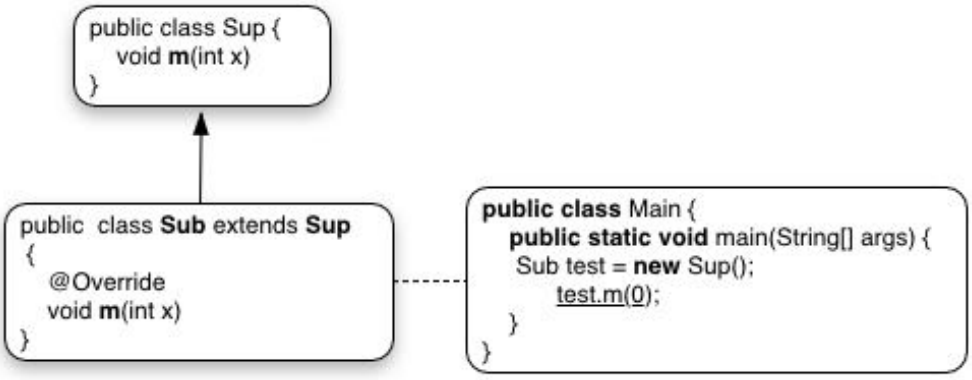


Figure 6.9: Renaming method override code

method from the *Sub* class for now.

Simplified CFD (Change Function Declaration) Mechanics steps:

- Use Extract Function on the function body to create the new function.
- Apply Inline Function to the old function.

Quoted from Fowler's Mechanics: *If you're changing a method on a class with polymorphism, you'll need to add indirection for each binding. If the method is polymorphic within a single class hierarchy, you only need the forwarding method on the superclass. If the polymorphism has no superclass link, then you'll need forwarding methods on each implementation class.*

In light of this CFD migration variant, the EF-refactoring consists of the following steps:

EF (Extract Function) Mechanics steps

- Create a new function, and name it after the intent of the function
- Copy the Whole body from the source function into the new target function.
- Replace the extracted code in the source function with a call to the target function.

We now proceed with or step described at What-If develop recipe (subsection 5.2.2). The identification of the Mechanics and mapping the action to Microsteps gives Figure 6.10.

A quick assessment of Risks shows us that many dangers are lurking in the case of the AM (Add Method) Microstep, as is known for renaming a method. See also here for related dangers listed: Rename Method What-If Use Cases (subsubsection 5.1.7). Again for demo purposes, we only care for the affected methods in the inheritance tree that should be renamed as well.

In that case, our "Overriding validation" What-If Figure 6.11 will be defined as:

```
IF method-overriding@m THEN output conclusion ...
```

Typically, the Identification of What-Ifs for method overriding through What-If develop recipe (subsection 5.2.2) results in more than applicable What-If, of course!

For each What-If, we have at least one detector: *detector:method-overriding?method-name*.

```
1 method_is_overriding(MethodId) :-
2   ground(MethodId),            % method is known
3   methodT(MethodId, ClassId, Name, Params, _, _, _, _), % method has
      params
4   subtype(ClassId,SuperClass), % enclosing class  has a superclass
5   type_contains_method(SuperClass,MethodId2), % superclass contains method
6     methodT(MethodId2, SuperClass, Name, Params2, _, _, _, _), % which
      also with params
```

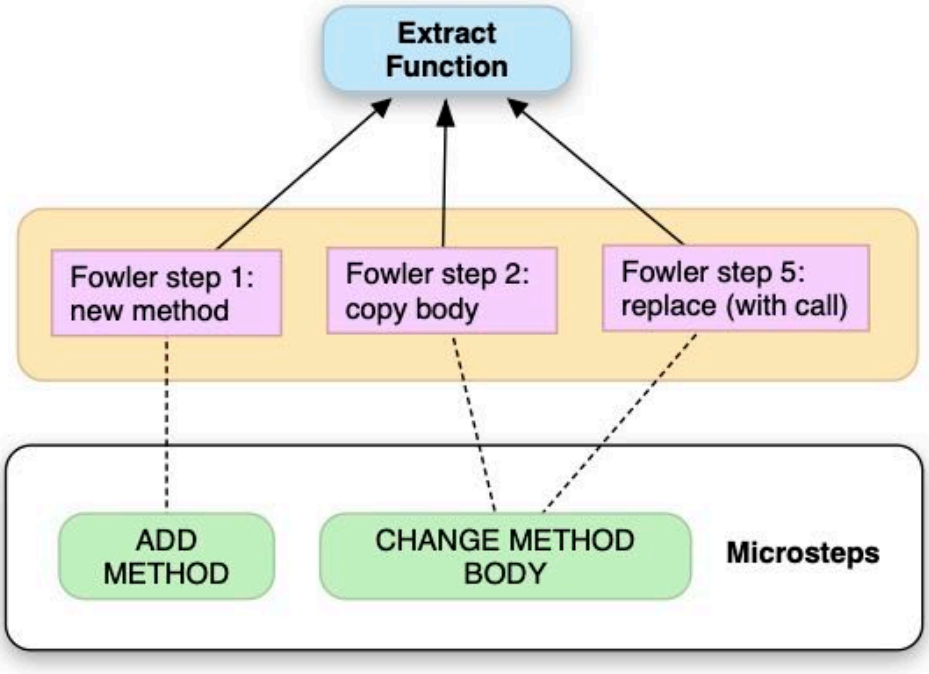


Figure 6.10: EF-refactoring related Microsteps

Figure 6.11: Renaming method override What-If

```
7   not(modifierT(_, MethodId2, 'abstract')), % overridden method not
      abstract
8   not(modifierT(_, MethodId2, 'private')),  % overridden method not
      private
9   equal_parameter_types(Params,Params2),    % params number and typing
      match
10  !.
```

Listing 6.2: Detector for overriding condition listing

6.3.2. EF-refactoring demo 2

Now let us make things a bit more complicating by switching to another source code project Figure 6.12, with a rather complex inheritance structure in comparison to Figure 6.9. In addition, we want to receive warnings for all affected abstract methods. With inheritance, regarding abstract method refactoring, we also have to watch out for the branch toward the top-level abstract method declaration and even those children. Because of this, our detectors will need to cope with this as well. We figured this out at Figure 5.5 already as part of the What-If to detector chaining discussion at Detector chaining concept (subsection 5.1.5).

Class B (colored blue) contains the to be renamed method *m(int)*. The (red colored) numbers indicate the detector number *det*. The detectors *det1*, *detS* match on method level. The detectors *detC*, *det2*, *det3*, *det4*, *det5* match on class level.

Tooling

In the tooling, we have divided the analysis into multiple parts. Here Figure 6.13, you can see both Detectors and What-Ifs as analysis entries. Execution of our What-If results in three matches. It proves that when we rename method *B::m(int)*, the corresponding classes found are: Classes A,B and D.

At any level, we can inspect the analysis results. As we see in Figure 6.12, our detectors 1–5 show the following analysis results.

Detector 1: *classes?@same-methods*
Detector 2: *superclasses?@class*
Detector 3: *abstract-superclasses?@superclasses*
Detector 4: *subclasses?@abstract-superclasses*
Detector 5: *matching-subclasses?@common classes*

Reasoning

Our Reasoning mechanism (subsubsection 5.3.2) has been reasoning about the outcome from detector *det3* with matching classes [A] and detector *det5* with matching classes [B,D]. Our What-if Figure 6.14 covers about affected methods *m(int)* for the classes [A,B,D] also requiring to be refactored. The upcoming code list shows the internals of our What-If. The outcome has been determined by taking the *union* of both branches.

```
1 det6_abstract_method_implementation(Classes) :-
2   det3_determine_abstract_superclasses(SuperclassSet),
3   det5_matching_children(Children),
4   union(SuperclassSet, Children, Classes).
```

Listing 6.3: Detector abstract-method listing

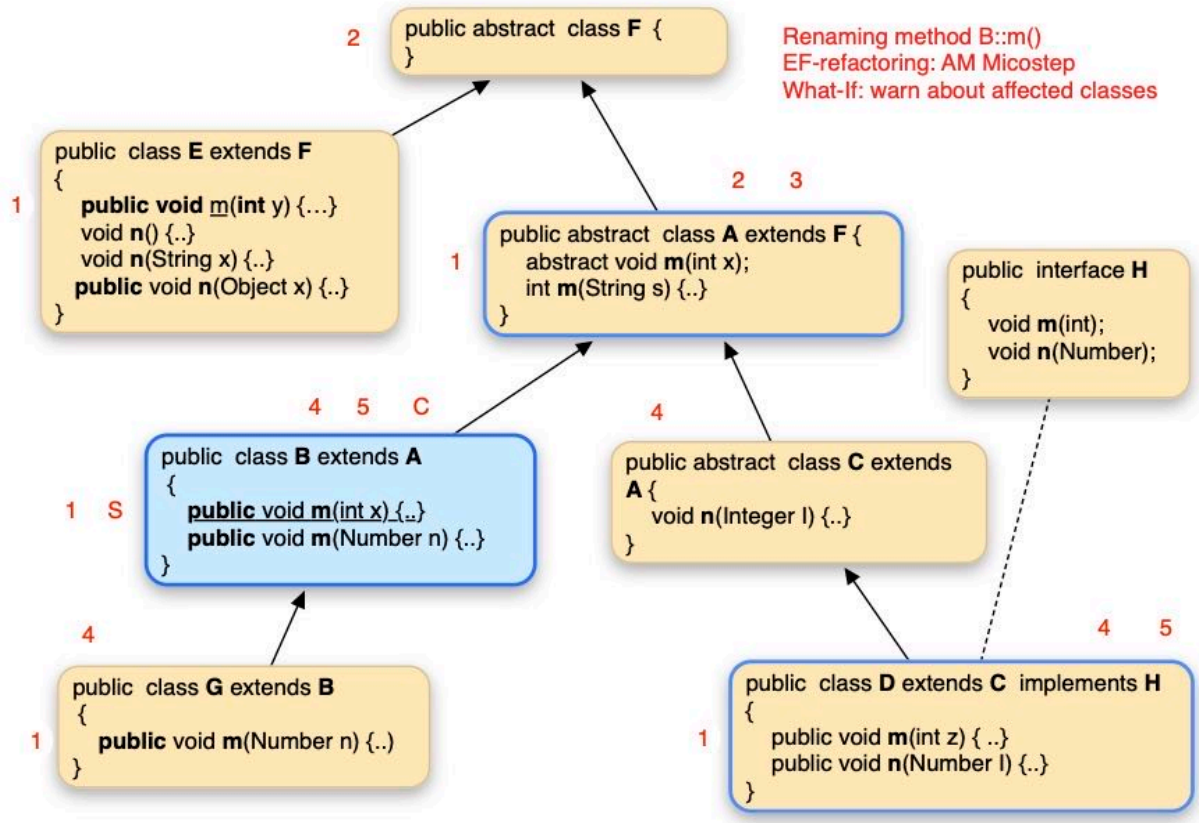


Figure 6.12: What-If execution, detector hits

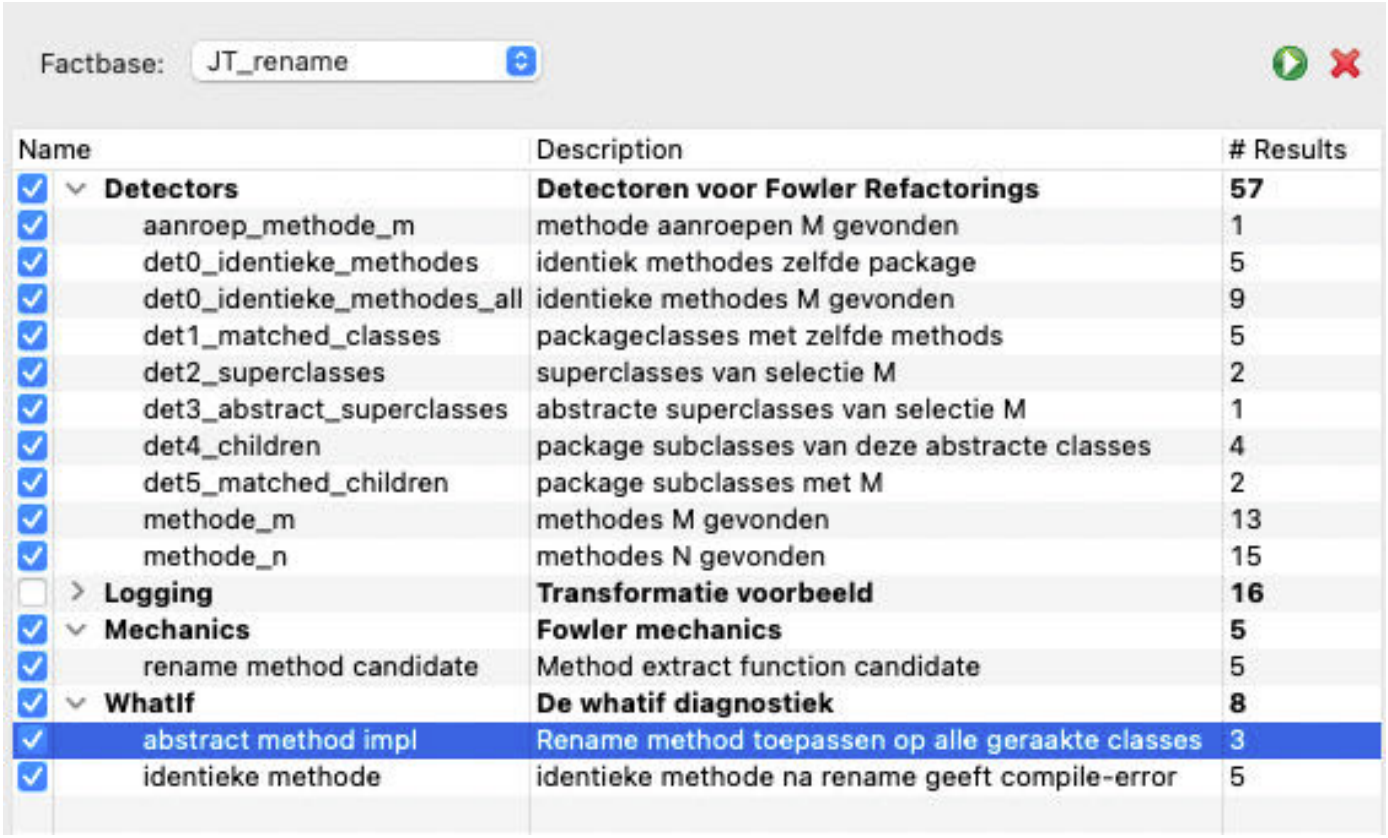


Figure 6.13: Tool console for the Rename Method project: WhatIf

Figure 6.14: What-If analysis result for Rename Method example

Figure 6.15: Detector 1: output matching classes with same methods

Figure 6.16: Detector 2: output matching superclasses for class

Figure 6.17: Detector 3: output matching abstract superclasses out of superclasses

Figure 6.18: Detector 4: output matching subclasses for abstract-superclasses

Figure 6.19: Detector 5: output for matched subclasses for common classes

7

Related Work

As elaborated in sections Refactoring Foundations (section 1.1) and Fowler Refactoring basics (section 1.3), the Fowler Refactoring Mechanics is our reference point.

The problem domain regarding our study is how to establish the foundations for Risk-Based Refactoring Guidance, based on Fowler's Refactorings. We are concerned about three areas of interest Figure 7.1 within the refactoring process that we want to address with our Framework for this study.

We could not find related work encompassing all intended elements of interest together:

- Refactoring guidance, allowing for active advice to alert the user on demand
- Refactoring risk prevention mechanism, code diagnostics, and analysis to reason about potential dangers
- Refactoring tooling support, integration into an IDE to support both refactoring and guidance

Guidance

In the area of applied learning, current OU-course lectors Sylvia Stuurman and Harrie Passier (along with former lector and fellow supervisor Lex Bijlsma) wrote an internally documented research plan: 'Software Quality in Education', based on Stuurman's thesis 'Design for Change' [Stuurman, 2015]. Explicit guidance is necessary to cater to the lack of systematic problem-solving strategies regarding the three perspectives: strategy, teaching, and tools. Those perspectives are intertwined, but no perspective may be left out, namely, no tools without good strategies that prescribe what kind of problems to solve and tools supporting the teaching within those strategy parameters.

As they summarized: "In the field of refactoring software, one of the items of interest to explore is if there is a positive effect on understanding programming when assist refactoring with explicit guidance".

Figure 7.1: Refactoring problem domain

Hence, as a spin-off, former OU-graduate Patrick de Beer started researching, what he coined as 'Refactor Guidance'. The works of Patrick (still involved with the project) has resulted in a prototype tooling that is (although in theory) able to supply contextual bits of advice bases on the detection of the code under investigation. He concluded in his study that none of the academic tools that assist a student in refactoring, like the mentioned JspIRIT[1] as a candidate, generate advice as feedback. Patrick has introduced a novel approach to risk-based refactoring.
Tom Mens [Mens and Tourwé, 2004] has defined a process for refactoring, but the part of risk-based refactoring was still missing. More information and a short introduction about the prototyped concepts defined by Patrick de Beer can be found in the Appendix at Refactor Guidance intro (Appendix E). Also, issues with his proposed RAGs (Refactoring Advice Graphs) can be found there.

The Intelligent Tutor System (ITS), described by Händler et al. [Haendler et al., 2019] with a focus on interactive tutoring software refactoring, states that their ITS provides "feedback to the users … regarding the software-design quality and the functional correctness of the (modified) source code". Main focus is on the architectural depth and the method they use is comparing UML as-is (by reverse-engineering the code) and the to-be situations with each other. Their proof-of-concept supports the notion of feedback, however, not from the perspective of giving advice about potential refactoring dangers and how to coop with them accordingly.

At this time of writing, in her Ph.D, Hieke Keuning [Keuning, 2020] made progress with an ITS-based solution for Automated Feedback for Learning Code Refactoring. To my knowledge, these code refactoring examples are limited to only code optimizations with method scope, for instance, the code refactoring for adding the sum of

[1]JspRIT (jan–2020): https://sites.google.com/site/santiagoavidal/projects/jspirit

values tutors replacing for loop with for-each loop.

Risk prevention
To measure the level of impact of changing code, Melina Mongiovi [Mongiovi et al., 2014] introduced the notion of SGTs (Small-Grained Transformations), similar to the concept of our Microstep level (section 4.1). They use SGTs only to identify the cause of impact to reduce the number of autogenerated test cases on behavior preservation breaks caused by these specific small-grained transformation. We, on the other hand, use Microsteps as a vehicle to derive our refactoring specific What-Ifs.

Tooling support
Nikolaos Tsantalis is the author of the JDeodorant[2] Eclipse tooling plug-in, and co-author of many refactoring-related articles about detecting code smells and automating detection and refactoring into tooling. His research thesis [Tsantalis, 2010] addresses integrating code smell detection functionality into the Eclipse IDE.

The MPS editor from Jetbrains[3] is an example IDE tool that supports AST and DSL processing out of the box. They have implemented tutorials to learn to refactor, but they lack the guidance advisory role.

In Future Work (chapter 9), we take a closer look at alternatives for parsing and querying code for detector implementations other than based on the traditional AST approach. Issues with refactoring in general and tooling-related support have been summed up in the appendices at Complexity of refactorings (section B.1) and Issues with tooling (section B.2).

[2]https://github.com/tsantalis/JDeodorant
[3]https://www.jetbrains.com/mps/

8

Conclusion

Main research question
To achieve the fulfillment of our main research question, "How can we **conceptualize** and deploy a system that delivers contextual-based refactoring **guidance**?", we developed the following concepts as part of our Refactoring Guidance Framework. Each underlying concept has been addressed and explained separately:

- Detector, Detector Chaining, Detector Layering, Code Diagnostics, What-If, Verdicting, Microsteps, Risk-based Advice
- Building Block Logic, Building Block Composition, Black-box Notation, Tooling Architecture

We accomplished building a prototype to conclude our concepts. The practical use of single tasked detectors, how they cooperate (detector chaining) in an orchestrated fashion (detector layering), and the unified construction (building block logic), prove well enough the underlying principles for risk-based refactoring guidance. We have been working out some examples to demonstrate our mentioned Framework concepts in a dedicated chapter, Prototyping the Framework (chapter 6).

Guidance improvement: The process of refactoring can be visualized as a flow of steps to be processed. We went further than the standard Fowler refactoring Mechanics. The addition we present is that we include the notion of risk. At the start of the refactoring processing flow, we already recognize risks and guide the refactoring practitioner with advice to prevent these risks. See Risk-based Refactoring Process (subsection 2.3.1) for more details.

Research question I
Regarding RQ1: "How can we define a framework for **decomposing** the Fowler refactoring Mechanics into composable actions to enable guided **instructions** on proceeding based on the actual **code context**?"

Our Framework for the decomposition of Fowler Refactoring to Microsteps covers RQ1 and is part of the discussion, topic wise at:

- Decomposition of Fowler refactorings: Mechanics level (subsection 1.3.1), Mechanics steps level (subsection 1.3.2), Microstep level (section 4.1), Matrix of Microsteps (section 4.2), Matrix properties (section 4.3)
- Guiding assistance: Risk-based Guidance Use Case (subsection 2.3.2)

Our contribution towards the refactoring process is twofold, we introduced the concept of Microsteps attached to the Fowler refactoring actions. Each Microstep has potential risks. With detectors we are able to detect whether a potential risk is a real risk in a certain code context.

Research question II
Regarding RQ2: "How can we devise a generic set of code context **detectors** for the selected refactorings that will enable the **tooling** to **assist** in guiding the student?"

With the aid of detectors, we can capture the code state for any given code context. By employing specialized detectors called 'What-Ifs', any involving risks can be reasoned about (verdict processing), and the resulting outcome of this reasoning is presented as advice (through template-based instructions).

For RQ2, we devised the necessary detector technology based on method-related refactoring examples (Change Function Declaration and Extract Function). We explain how to obtain and set up detector logic, the variety of detectors, their interaction, and how to employ these detector chains to make guiding assistance possible. What-Ifs facilitate to assist the students before actual refactoring takes place. We utilized a particular purpose black-box notation suited for describing the detector's tasks. Our detector chaining concept greatly benefits from this notation manner as it helps providing a helicopter overview over the participating detector interaction.

Please refer for further reading:

- Detectors and chaining of detectors: Detectors (section 5.1), Detector chaining concept (subsection 5.1.5), Detector compositions (subsection 5.1.3)
- Tooling: Prototyping the Framework (chapter 6)

As a showcase for the assistance of students, we managed to set up a prototype tooling to implement the construction of a detector chain. The tooling is based on interchangeable layered based architecture. Restriction for the prototype tooling is that we currently support feed-forward advice only, meaning that the user requests advice before actual code changes.

Research question III
Regarding RQ3: "How can we devise assisting the student with **advice** by reasoning about the conditions to be met for accurate refactoring **diagnostics**?"

Main focus for RQ3 is the applied technique to deliver advice. The quality of the

resulting advice depends on the gathered code diagnostics. Accurate code diagnostics are necessary for making good decisions on dealing with the possible hazards when applying the refactoring actions.

Our addition is the What-If concept to bridge code diagnostics and advice through the Verdict process (subsection 5.3.1). Each What-If constitutes an IF part for case based matching conditions and a THEN part to reason about which advice or remedy to deliver to the student.

Reasoning about code state and the deliverance of advice has been covered by:

- Reasoning and Advice: Reasoning about dangers (subsection 2.3.3), What-Ifs (section 5.2), Verdict Idea (section 5.3)
- Code diagnostics: Source code Diagnostics (chapter 5)

Currently, the approach of reasoning with the aid of Arbiter/Expert Opinion Tables has not yet been crystallized out and has room for debate. However, as explained in Detector Building Blocks concept (subsubsection 5.1.3), the standard verdict combining logic is proficient for the task. On many occasions, it is sufficient to use logical operators to handle the many outputs from detectors. For example, the union operator merges the results between detectors as we did in our prototype.

Research question IV
Reqarding RQ4: "How can we integrate **monitoring** functionality into the toolset to assess the execution of the manual refactoring steps?"

Initially, at the start of the assignment, we opted that tracking the progress of a refactoring fits the Risk-based Refactoring Process (subsection 2.3.1), hence the research question. However, gradually, RQ4 appeared way out of scope, also given the duration of the assignment. So we concluded that monitoring should be regarded as Future Work (chapter 9).

9

Future Work

The scope of the study can be extended in the field of refactoring guidance techniques and tooling support.

I) Refactoring guidance related recommendations
a) Coverage of Fowler Refactorings:
From the perspective of usability, better coverage of the number of refactorings from the Fowler catalog is recommended. For example, covering the top 3-listed refactorings, as noted by Patrick de Beer in his thesis [de Beer, 2019], will drastically improve usability. This goes along with realizing more What-Ifs and perhaps detectors when not reusable already. Besides the quantity of What-Ifs, we also need to evaluate about those non functional quality aspects such as reusability. A non-exhausting list of What-Ifs, to accomplish or improve coverage for the Rename Method refactoring, is available here: Table 5.7 and Table 5.8.

b) Architectural Decay prevention:
Fowler's refactoring Mechanics guarantee not to break behavior preservation. However, OO-language concepts contribute as usual suspects to breaking behavior preservation and are primary sources for our What-Ifs.
Joshua Kerievsky [Kerievsky, 2005] introduced the *Refactoring to Patterns* paradigm to overcome the difficulties and common pitfalls related to this kind of refactoring. Suppose a student attempts to refactor the structure of a Design Pattern. In that case, we probably do not have the appropriate What-If in place to warn us about risks involving architectural decay. As Jason McC. Smith [Smith, 2012] explains the decay with an example renaming action breaking the composition of the Decorator Design Pattern. Checking the Design Pattern's integrity on the elementary level of EDPs may find signs of architectural decay. EDPs (Elemental Design Pattern) are the building blocks of the Design Patterns. As a remedy, we recommend designing EDP compliant What-Ifs and accompanying detectors.

c) Further research is needed to practice the concepts of source code diagnostics, the identification and functionality of What-Ifs, or the introduction of cascaded verdicts based on Multi-Criteria Decision Analysis. see: Cascaded Verdict (section A.3).

d) In the previous Conclusion (chapter 8) chapter, we decided why RQ4 (monitoring progress) is Future Work (chapter 9).

II) Tooling enhancements
a) Track and trace refactoring support:
Continuous refactoring guidance by monitoring and measuring the progress of the refactoring is key to enhancing support for feedback besides feedforward. For example, in the current prototype solution, employing the JTranformer AST transformation capabilities facilitates watching the effects of a Microstep.

b) Composing detector-What-If chains, with the aid of low-code solution utilities:
Similar to the idea of Software Product Lines, where we can assemble products out of individual components forging them together, this might also be applicable to designing and maintaining our detector chains.
We could even think of doing this visually, conducted by diagramming tools such as Eclipse Papyrus or OpenPonk.

III) Tooling alternatives
a) Investigating AST alternatives:
The built-in AST implementation of Eclipse is much too heavy-weight to support for a long time. Our suggestion is to abstract away from direct use of the Eclipse JDT APIs. Instead of JDT, some of the following alternatives might be interesting, like MPS with its DSL support, Famix AST modeling (Pharo language), Rascal M3, Java Spoon Framework, or one of the many more mentioned at section A.2 in the appendix.

b) IDE Hosting platform alternatives:
Deploying on other IDEs like Jetbrains IntelliJ, Visual Studio Code, Netbeans IDE, or Jetbrains MPS instead of Eclipse might reach a broader or even other audience otherwise. Other IDEs may also offer other feature possibilities.

c) Alternatives for querying the state of code:
The AST and the language to query the AST, as used in detectors, can be varied by other techniques (besides our proposal described at Conceptualizing the AST):

- Java Code Ontology adoption (based on OWL 2.0) to specify yet another division of an AST
- Java source-code context as RDF triplet datasets and SPARQL as the host query language (conform Java Code Ontology specs?)
- Graph-based databasing as a representation of the code's AST; for example, Neo4J is a competent property graph database to build and query the AST. There is native support for the Cypher query language along with many client language APIs. Third-party solutions such as JQAssist support the transition from Java to Neo4J node based AST as well
- XPath-related AST, custom-built solution (with ANTLRv4?)

In the Appendix Ideas for further research (Appendix A) we point to more information and interesting articles to read for future work.

Source references:

- https://www.jetbrains.com/mps/
- http://codeontology.org
- https://yasgui.triply.cc
- https://jqassistant.org
- https://pharo.org
- https://openponk.org
- https://sewiki.iai.uni-bonn.de/research/jtransformer/start

Bibliography

B

Jan Bečička, Petr Zajac, and Petr Hřebejk. Using Java 6 compiler as a refactoring and an analysis engine. *Suites. The TRex TTCN-3 Refactoring and Metrics Tool..... 3*, (The TRex TTCN-3 Refactoring and Metrics Tool):56, 2007. 27, xv

Kent Beck. Embracing Change with Extreme Programming. *Computer*, 32:70–77, 1999. ISSN 0018-9162. doi: doi.ieeecomputersociety.org/10.1109/2.796139. 3

Patrick de Beer. *Code Context Based Generation of Refactoring Guidance*. MSc thesis, Open University, 2019. 2, 17, 50, 111, xxix

C

Márcio Cornélio, Ana Cavalcanti, and Augusto Sampaio. Sound refactorings. *Science of Computer Programming*, 75(3):106–133, 2010. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2009.10.001. URL http://www.sciencedirect.com/science/article/pii/S0167642309001300. xiv

F

Martin Fowler. *Refactoring: Improving the Design of Existing Code 2nd Ed., 2018*. Addison-Wesley Professional, 2018. 1, 2, xxvi

G

William G Griswold and William F Opdyke. The birth of refactoring: A retrospective on the nature of high-impact software engineering research. *IEEE Software*, 32(6): 30–38, 2015. 2

H

Thorsten Haendler, Gustaf Neumann, and Fiodor Smirnov. An interactive tutoring system for training software refactoring. *Instructor*, 1:4, 2019. 106

K

Joshua Kerievsky. *Refactoring to Patterns*. Pearson Deutschland GmbH, 2005. 2, 111, x, xv

Hieke Keuning. *Automated Feedback for Learning Code Refactoring. Open Universiteit*. PhD thesis, Open University, September 2020. URL https://research.ou.nl/en/publications/automated-feedback-for-learning-code-refactoring. 106

Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. Improving refactoring speed by 10x. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1145–1156. IEEE, 2016. xiv

M

Tom Mens. On the use of graph transformations for model refactoring. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 219–257. Springer, 2005. xiv

Tom Mens and Tom Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.1265817. URL http://dx.doi.org/10.1109/TSE.2004.1265817. cites: beck_test_2002 cites: mens_survey_2004. 17, 106

Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005. 35, xiv

Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. Making refactoring safer through impact analysis, 2014. *Science of Computer Programming*, 93:39–64, 2014. doi: 10.1016/j.scico.2013.11.001. 30, 107

Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. Detecting overly strong preconditions in refactoring engines. *IEEE Transactions on Software Engineering*, 44(5):429–452, 2017. xiii

Emerson Murphy-Hill. Programmer friendly refactoring tools. 2009. xv

O

Jonhnanthan Oliveira, Rohit Gheyi, Melina Mongiovi, Gustavo Soares, Márcio Ribeiro, and Alessandro Garcia. Revisiting the refactoring mechanics. *Information and Software Technology*, 110:136–138, 2019. xvi

Jeffrey L Overbey, Ralph E Johnson, and Munawar Hafiz. Differential precondition checking: A language-independent, reusable analysis for refactoring engines. *Automated Software Engineering*, 23(1):77–104, 2016. xiv

S

Max Schaefer and Oege De Moor. Specifying and implementing refactorings. In *ACM Sigplan Notices*, volume 45, pages 286–301. ACM, 2010. x

Max Schäfer, Torbjörn Ekman, and Oege De Moor. Sound and extensible renaming for Java. In *ACM Sigplan Notices*, volume 43, pages 277–294. ACM, 2008. xiv

Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. A comprehensive approach to naming and accessibility in refactoring Java programs. *IEEE Transactions on Software Engineering*, 38(6):1233–1257, 2012. xiv

Jason McC Smith. *Elemental Design Patterns, 2012*. Addison-Wesley, 2012. ISBN 978-0-13-4844354-4. 25, 63, 111, x

Gustavo Soares, Melina Mongiovi, and Rohit Gheyi. Identifying overly strong conditions in refactoring implementations. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 173–182. IEEE, 2011. xiii

Friedrich Steimann. Constraint-based refactoring. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(1):2, 2018. xiv

Sylvia Stuurman. *Design for Change*. PhD thesis, Open Universiteit, Heerlen, June 2015. 105

T

Andreas Thies. *Constraintbasierte Refaktorisierung von Deklarationen in JAVA*. PhD thesis, 2014. xiv

Nikolaos Tsantalis. Evaluation and improvement of software architecture: Identification of design problems in object-oriented systems and resolution through refactorings. *Diss. Ph. D. dissertation, Univ. of Macedonia*, 2010. 107

V

Mathieu Verbaere, Ran Ettinger, and Oege De Moor. JunGL: A scripting language for refactoring. In *Proceedings of the 28th International Conference on Software Engineering*, pages 172–181. ACM, 2006. xv, xxiv

Listings

Alphabetical Index

Acronyms

AST Abstract Syntax Tree.

DAG Directed Acyclic Graph.

EDP Elemental Design Pattern.

EOT Expert Opinion Table.

IDE Integrated Development Environment.

ITS Intelligent Tutoring System.

OWL Web Ontology Language.

RAG Refactoring Advice Graph.

RDF Resource Description Framework.

REPL Read-Eval-Print-Loop.

REST REpresentational State Transfer.

SGT Small-Grained Transformation.

TAT Template Advice Table.

Glossary

AST In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language. Each node of the tree denotes a construct occurring in the text..

HTTP method The HTTP method specifies the intention of a HTTP request. The most common types of request are: GET, HEAD, PUT, DELETE and POST.

plug-in The plug-in architectural style allows applications to be extended by providing an interface. Plug-ins can implement this interface to provide custom functionality.

Appendices

A

Ideas for further research

A.1. Refining refactorings

The Fowler refactoring catalog is a good source for implementing our Use Cases, Another good source is that of Joshua Kerievsky's "Refactoring to Patterns" [Kerievsky, 2005], has he explored procedure to refactor towards Design Patterns. In his book he gives numerous examples of how to perform the individual mechanic steps. Even further, Jason McColm Smith's book about "Elementary Design Patterns" [Smith, 2012], talks about these EDPs as building blocks on which Design Patterns are constructed upon, but with a strong dependency to mechanisms we can use to learn about method operation transformations.

Besides the elementary building blocks for Design Patterns. McC Smith plotted all method relationship into a three dimensional cube. This cube has the following properties: object (dis)similarity, type (dis)similarity and method (dis)similarity. Method operations are formalized as ρ-calculus reliance operators, based on OO-conceptual σ-calculus based on procedural λ-calculus.

The following illustration EDP method call cases (Figure A.1) shows all different method calls possible for the Java language derived from the method reliance operators.

An interesting article about micro-refactorings can be read in an article from OOPSLA 2010 with proposed refactoring decomposition techniques to get to micro-refactorings by Max Schäfer and Oege de Moor [Schaefer and De Moor, 2010].

A.2. Related refactoring tooling and articles

The following tooling is available on the market that can perform Java code refactoring and are interesting to look at. The articles depicted in the figure and published articles from listed authors cover a broad overview of the many related issues addressed in this document such as those briefly described in the problem context.

IDE tooling One of the two well known Java development environments are IntelliJ and Eclispse. Both support entry level refactoring and can be extended by plug-

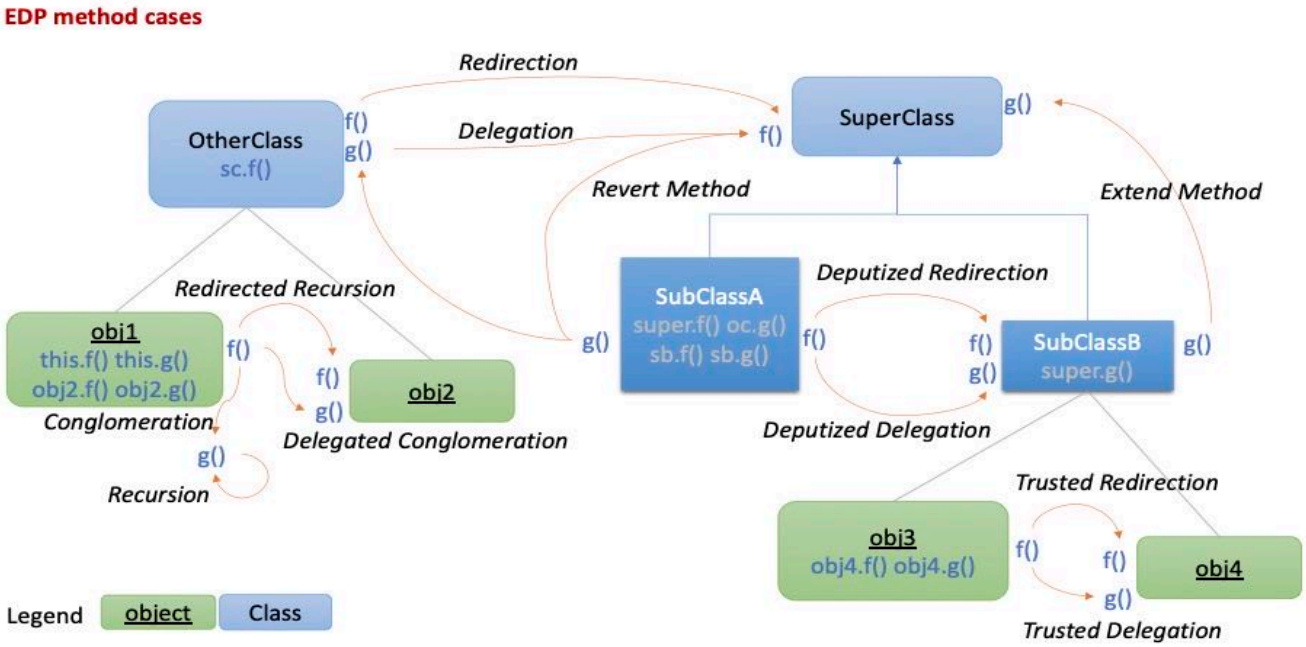


Figure A.1: EDP method call cases

ins, like: J*Deodorant , JspIRIT, jSparrow, RefactoringMiner, JMove, Stench Blossom, TopicViewer, iPlasma, FrenchPress, AutoStyle, SonarQube, MPS, PMD and Checkstyle.*

AST or Constraint based tooling Throughout all the articles scanned the following mentioned software can aid in analyzing and manipulating the Abstract Syntax Tree: *ROOL, jastAdd, Maud,Z, SafeRefactor, SaferRefactoring, Alloy, ASTGen, Coq, Refacola, CafeOBj, rCOS, JaMoPP, JavaParser, JDT, Spoon, Moose, MPS and Rascal.*

Influential authors regarding refactoring Key players who were and still are essential to refactoring : Marin Fowler, William Griswold, William Opdyke, Robert C. Martin, Scott Ambler, Joshua Kerievsky and Kent Beck.

The following extensive list are names of researchers who play or played an influential role in the domain of software refactoring, in order of appearance of their first publication: Márcio Cornélio, Günter Kniesel, Helge Koch, **Tom Mens**, Ana Cavalcanti, Augusto Sampaio, Alejandra Garrido, José Meseguer, Mathieu Verbaere, Ran Ettinger, **Emerson Murphy-Hill**, Oege de Moor, **Max Schäfer**, Torbjörn Ekman, Johan Åkesson, Torbjörn Ekman, Görel Hedin, Hannes Kegel, Friedrich Steimann, Nicolas Juillerat, Quinten David Soetens, Gustavo Soares, **Nikolaos Tsantalis**, Jeffrey L. Overbey, Friedrich Steimann, Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh. Rajkumar, Brian P. Bailey, Ralph E. Johnson, Andreas Thies, Frank Tip, Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, Paulo Borba, Erland Kristiansen, Anna Maria Eilertsen, Jongwook Kim, Don Batory, Danny Dig and Maider Azanza.

The figure Figure A.2 is a subset of the articles from the mentioned authors and some arrows between them to indicate a citation or cite quotation. Most of the articles within the figure are candidate study material.

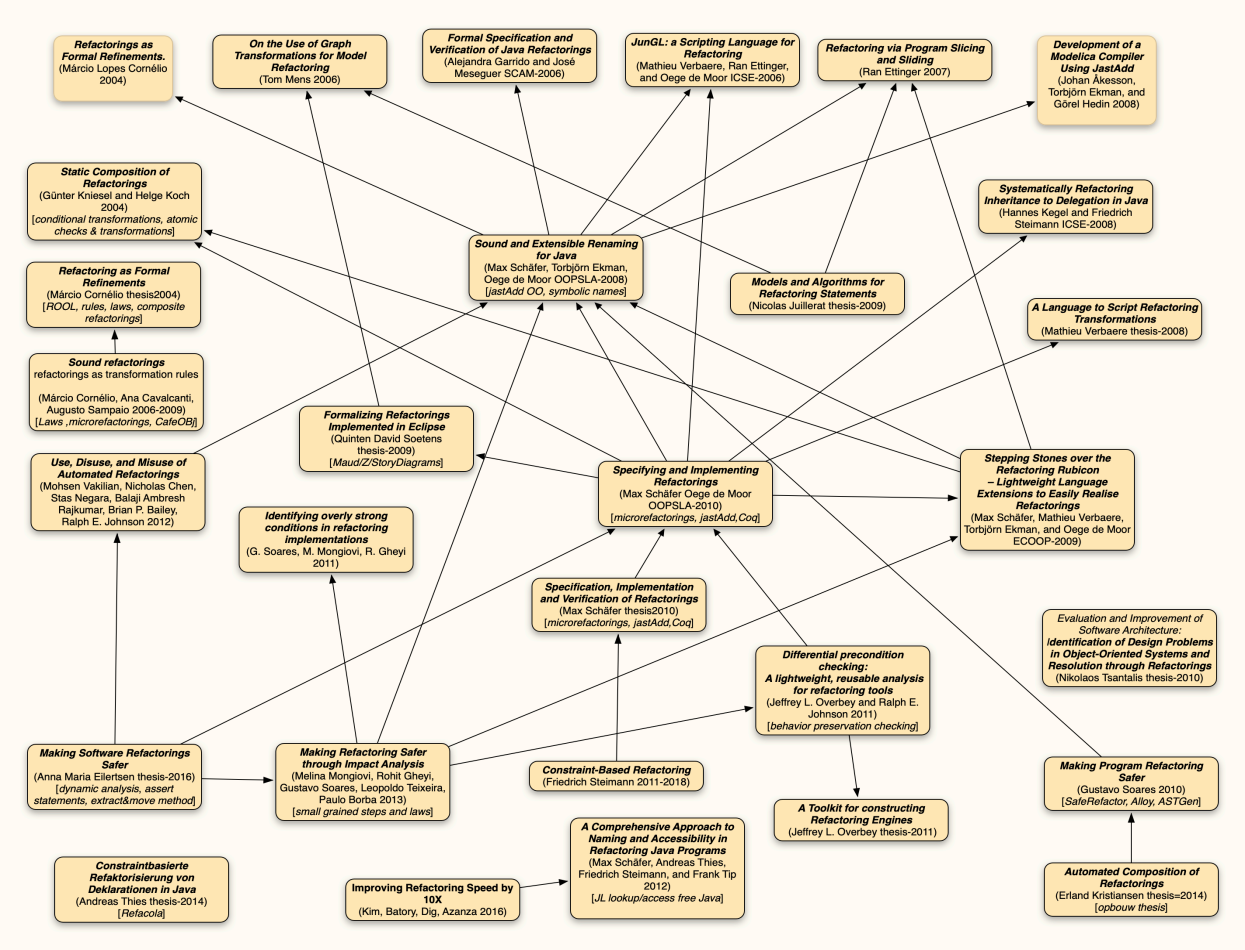


Figure A.2: Refactoring articles citing others

A.3. Cascaded verdict

The idea of a weighted verdict can be elevated even further with the introduction of the level of importance/relevance on cascading What-Ifs.

Our suggested technique for decision making is to weight the inputs of the underlying What-Ifs by means of applying the Weighted sum formula. This technique of decision analysis is based on Multi-Criteria Decision Analysis (MCDA[1]) to evaluate Model-based Product Quality[2].

Weighted Sum Formula

$$WI_A = \frac{\Sigma(WI_{a,i} * WI_{RL,i})}{\Sigma(WI_{RL,i})}$$

[1]MCDA Multi Criteria Decision Analyses: Model–based Product Quality Evaluation with Multi–Criteria Decision Analysis*1401.1913, 2010*11*Metrikon*02*authors*version

[2]book: Software Product Quality Contro from Stefan Wagner

Here, *WI* stands for What-If, with A indicating our arbiter What-If-A in case we have more than one arbiters. *WI-a,i* refer to the Arbiters dependent Child's verdict weight level (Verdict weight levels (subsubsection 5.3.2)) . Each dependent child *WI-a,i*, indexed by *i* has a relevance level value *WI-rl,i* with value ranging RL=[1–3, 1=least important till 3=most important].

The outcome of the Formula must be rounded down to whole number.

The outcome of the formula is used in the Arbiter What-If's verdict logic. Note that the Arbiter What-If likewise can lookup its own verdict weight value as well, comparing this to the Weighted Value evolution formula outcome, how to verdict, or simply ignore the evaluation.

A practical application for having an arbiter kind of What-If would be in the case when we consecutively execute both Remove - and Add Microsteps in order to modify some source code. For us it is obvious that the Remove action introduces a temporary state fixed by the Add action, but the attached What-Ifs arguably will produce ill-intentioned advice. The verdict engine's task is to prevent this.
Fowler refactorings concerning inheritance (such as Pull Up Method) are candidate to include an Arbiter What-If.

Example calculation
Suppose that between the range [1–5] the weighted value function for this step returns 4.

Main What-If A depends on the supporting What-Ifs A1 and A2. Given this knowledge, Main What-If A can calculate the weighted value for each of the underlying What-Ifs, can be obtained by simply calling the same weighted value function on their behalves again.

For instance, this gives us the following weight elicitation: A1 = 3, A2 = 5

Relevance Levels [1=least important till 3=most important] : A1 = 3 , A2 =1, which gives: (A1 * RL-A1) + (A2 * RL-A2). /(RL-A1 +RL-A2) gives (9 + 5) /4 = 3.5 rounded down gives 3

This outcome indicates that the advice of A2 can be repeated, but A1 can be superseded by Main-If A's own interpretation

A.4. Detector optimalisations

Some novel techniques that are described in published articles and maybe interesting for use are articles about overly strong preconditions from Gustavo Soares [Soares et al., 2011] and Melina Mongiovi [Mongiovi et al., 2017]. Overly Strong Preconditions may lead to wrong detection negatives. This means that the negative outcome prevents the refactoring execution on beforehand while in practice only little code adjustment is needed to succeed. So relaxing strong precondition to a kind of weaker

condition could overcome this. Naming and Accessibility can lead to very complex side effects.

Max Schäfer et al. [Schäfer et al., 2012] suggest to circumvent the side effects with an access free Java language rewrite. Detectors then have much less exceptions to cope with. At a later stage, specialized rules are in place to deal with a rewrite to the original differences.

A.5. AST optimizations

Precondition checking done by detectors for all situations can become quite cumbersome and overwhelmingly complex. To cope with complex transformations a technique called Differential Precondition Checking [Overbey et al., 2016] discussed in an article from Jeffrey Overbey, comes to aid. If we know on beforehand that we need to check on name binding, we should build ASTs that comprehend the *def-use* chaining of variables. With differential checking we can tell if changes in the AST are expected at certain locations in the tree because of name binding changes or because of refactoring mistakes.

An article from Kim battery and Dig Azanza [Kim et al., 2016] developed some techniques to increase the speed of AST querying techniques by using database technology for this. Tom Mens [Mens, 2005] [Mens et al., 2005] proposes the use of graph transformations for model refactoring in his articles.

A.6. Formal language laws

Several articles refer to the approach of refactorings as transformation rules that can be described in formal language laws. Márcio Cornélio et al. [Cornélio et al., 2010] compares these laws to micro-refactorings. Max Schäfer et al. [Schäfer et al., 2008] uses formal language and symbolic names to introduce sound and extensible renaming for Java programs.

Another approach that incorporates formal language is constraint-based refactoring. Andreas Thies introduces the Refacola engine in his thesis [Thies, 2014]. Friedrich Steimann [Steimann, 2018] also wrote an article about how to apply constraints as a mechanism for refactoring.

B

Refactoring issues

B.1. Complexity of refactorings

When the code base is (too) large or the refactoring becomes too complex, refactoring can become quite cumbersome. Especially novice users may not understand the intricacies of a complex refactoring and therefore not willing to let the tool do the job. On the other side, current refactoring tooling seems to be more targeted towards professional audience who already understand and are able to pinpoint the implications for a refactoring [Bečička et al., 2007].

It also may not be obvious to the inexperienced user that a particular refactoring is lowering the complexity. For example, refactoring towards a Design Pattern like the Composite pattern tends to increase the perceived complexity of the code [Kerievsky, 2005] for users that do not understand Design patterns or do not know how to refactor towards patterns.

B.2. Issues with tooling

Acceptance of refactoring tools Programmers may experience a barrier when they want to refactor with the help of tooling. Emerson Murphy-Hill [Murphy-Hill, 2009] observed from a survey among Agile Open congress participators, that more than half of the respondents answered that the tool is not flexible enough; it doesn't do what the programmers want. A quarter of the responses admit they have a lack of knowledge of the tool. Only a small percentage of the population answered they have a lack of trust in the refactor tool. Quality is another issue; as Emerson Murphy-Hill states, they have a good reason to distrust the correctness of the refactoring tool.

Refactoring tools are buggy Although refactoring tools, in general, do a pretty job in simple case refactoring, they still contain (even today, considering the reported issues in the bug tracking lists) a fair amount of bugs. Murphy-Hill refers to Verbaere [Verbaere et al., 2006] who has exposed several bugs, found in several refactoring tools, unfortunately changing behavior preservation. There is numerous documented evidence available about tooling that messes up the semantics of the original code after a refactoring operation. Patrick de Beer in his thesis [**?**] demonstrates an

example of a different outcome for the three popular IDEs: Eclipse, Netbeans and IntelliJ, even simple refactorings went wrong. A survey [Oliveira et al., 2019] among 107 respondents yielded that the generated output by such tooling is different from what they wanted.

Inconvenient use of refactoring tools Lack of knowledge of the tool, and perhaps lack of knowledge about refactoring in general, is a matter of getting educated in refactoring and to become acquainted with a preferred tool in mind. Why people tend to ignore refactor tooling or are reluctant to use a refactoring tool, could be because they find them inconvenient to use. After the execution, you only see the final result of the automated refactoring. No explanation about the reasoning of how the refactoring tool comes to its result. This does not contribute to the student's refactoring learning curve. The Refactor Guidance prototype tool from Patrick de Beer is an effort to address the lack of feedback.

Current state of existing refactoring tools To automate all the complex steps involved with the combined guidance and monitoring functionality, the tooling must be very sophisticated. I am not aware of tooling that can assist the student with all the single steps of the refactoring. And not a single refactoring tool is equipped to guide and monitor the refactoring. Neither there is a tool on the market that is proficient enough in doing simple till moderate refactorings flawlessly. Automatic refactoring implies that tooling, depending on the code context, make automatic decisions, sometimes for the worst. This does contribute to distrusting the tool.

B.3. Refactor Guidance RAG issues

The following issues have been identified for the Refactor Guidance tooling from our fellow researcher Patrick de Beer [**?**] .

- Proof of theory, still questionable if RAGs are ready for the task; According to de Beer "verification has been done with a limited number of scenarios and has not been used in real-life case scenarios".
- Limited functionality of the prototype, leads to RAGSs that are unnecessarily complex and inflexible;
- RAG implementation, The RAG is composed as a direct acyclic graph structure. Outgoing edge traversal is deterministic. The graph is further limited by an imposed restriction on the edges because outgoing edges they must be mutual exclusive. This makes it difficult to design RAGs
- **Controlling the order of execution**. Unfortunately the consequence of empty vertex introduction unwillingly exposes an anomaly that the same (edge-)labeling does not lead to the same advice.

C

Prototype code listings

This chapter contains library and detector sources used by the prototype demo.

C.1. detector sources

```
1
2 %%  start:  ------  method override listing   -------------
    method_is_overriding(MethodId) :-
3   ground(MethodId),                                % method is known
4   methodT(MethodId, ClassId, Name, Params, _, _, _, _),   % method has
     params
5   subtype(ClassId,SuperClass),                     % enclosing class  has a
     superclass
6   type_contains_method(SuperClass,MethodId2),    % superclass contains
     method
7     methodT(MethodId2, SuperClass, Name, Params2, _, _, _, _),  % with
     params
8   not(modifierT(_, MethodId2, 'abstract')),      % overridden method not
     abstract
9   not(modifierT(_, MethodId2, 'private')),       % overridden method not
     private
10  equal_parameter_types(Params,Params2),         % params number and typing
      match
11  !.
```

Listing C.1: method override listing

```
1
2 %%  start:  ------  Detectors for abstract class What-If   -------------:-
     module(detectors,
3    [  selector_method/3, selector_class/1,
4       det0_matching_methods/1, det0_matching_methods/2,
    det0_matching_methods_scoped/1,
5       det1_matching_classes/1, det1_matching_classes_scoped/1,
6       det2_determine_superclasses/1,
7       det3_determine_abstract_superclasses/1,
8       det4_determine_children/1,
9       det5_matching_children/1,
10      det6_abstract_method_implementation/1
```

```
   ]).

:- use_module('detectors').
:- use_module(library(lists)).

%% selector_method(-Method, -Newname, -InPackage)
%   detS - de detector die de refactor operatie aanduidt
selector_method(Method, Newname, InPackage) :-
    rename_selectie(Method, Newname,InPackage).

%% selector_class(-Class)
%   detC - detector voor bepaling class van selectie context
selector_class(Class) :-
  selector_method(Method,_,_),
  encl_class_or_self(Method, Class).

%% det0_matching_methods(-MethodList)
% matching methods met scope ALL
det0_matching_methods(MethodList) :-
    selector_method(InputMethod, _, _),
    findall(MethodDefs, same_signature(InputMethod, MethodDefs), Methods),
    MethodList = [InputMethod|Methods].

%% det0_matching_methods(-MethodList, ?PackageId)
% Matching methods met scope afhankelijk van aanwezigheid package van
   selector
det0_matching_methods(MethodList, PackageId) :-
  selector_method(InputMethod, _, PackageId),
  det0_matching_methods(Methods),
    findall( Method, (member(Method,Methods), encl_package(Method,
   PackageId)),MethodsInPackage),
    MethodList = [InputMethod|MethodsInPackage].

%% det0_matching_methods_scoped(-MethodList)
%  Maching methods met als vaste scope het package van de refactoring
   selector
det0_matching_methods_scoped(MethodList) :-
    selector_method(InputMethod, _, Package),
    findall(MethodDefs, same_signature(InputMethod, MethodDefs,Package),
   Methods),
    MethodList = [InputMethod|Methods].  %%append(InputMethod, Methods,
   MethodList) .


%% det1_matching_classes(-ClassSet)
% Geef alle classes waartoe de methods behoren
det1_matching_classes(ClassSet) :-
    det0_matching_methods(MethodList,_), %de auto scoped versie aanroepen
    findall(Cls,( member(M,MethodList), methodT(M, Cls, _, _, _, _, _, _)
   ), ClassList),
    sort(ClassList, ClassSet) .  %sorteert en filtert eveneens dubbelen

%% det1_matching_classes_scoped(-ClassSet)
% scoped variant van det1_matching_classes
det1_matching_classes_scoped(ClassSet) :-
    det0_matching_methods_scoped(MethodList),
```

```
61      findall(Cls,( member(M,MethodList), methodT(M, Cls, _, _, _, _, _, _)
        ), ClassList),
62      sort(ClassList, ClassSet) .  %sorteert en filtert eveneens dubbelen
63
64  %% det2_determine_superclasses(-ClassSet)
65  % Van de method enclosing class bepaal bovenliggende superclasses (
        transitief)
66  det2_determine_superclasses(ClassSet) :-
67    selector_class(Class),
68    findall(Superclass, (extends_recursively(Class,Superclass), sourceClass(
        Superclass)), ClassList),
69    sort(ClassList, ClassSet).
70
71
72  det3_determine_abstract_superclasses(SuperclassSet) :-
73    det1_matching_classes(PossibleClasses),
74    det2_determine_superclasses(PossibleSuperClasses),
75    intersection(PossibleClasses, PossibleSuperClasses, Candidates),
76      % de mogelijke kandidaten hebben dezelfde methodnaam en signatuur,
        maar is het Type abstract
77      findall(C, (member(C,Candidates), abstract_type(C)), SuperclassSet).
78
79  det4_determine_children(Children) :-
80    det3_determine_abstract_superclasses(SuperclassSet),
81    findall(T,( member(Cls, SuperclassSet), proper_subtype_recursive(T,Cls))
        , ClassList),
82    sort(ClassList, Children).
83
84  det5_matching_children(Children) :-
85    det4_determine_children(PossibleChildren),
86      det1_matching_classes(PossibleClasses),
87      intersection(PossibleChildren, PossibleClasses, Children).
88
89  det6_abstract_method_implementation(Classes) :-
90    det3_determine_abstract_superclasses(SuperclassSet),
91    det5_matching_children(Children),
92    union(SuperclassSet, Children, Classes).
```

Listing C.2: Detectors for abstract class What-If

C.2. library source

```
1
2  %%  start:  ------  inheritance   -------------:- module(inheritance,
       [equal_parameter_types/2, same_signature/2, same_signature/3,
3       subtype_recursive/2, proper_subtype_recursive/2,
4       extends_recursively/2,
5       abstract_method/1, abstract_type/1
6       ]).
7
8  %Warning:    Clauses of inheritance:extends_recursively/2 are not together
        in the source-file
9  %Warning:    Earlier definition at /Users/hermanhilberink/eclipse-
       workspace2/JT_rename_Prolog/pl/library/inheritance.pl:56
10 %Warning:    Current predicate: inheritance:'$pldoc'/4
```

```
%Warning:    Use :- discontiguous inheritance:extends_recursively/2. to suppress this message
:- discontiguous inheritance:extends_recursively/2.

%% subtype_recursive(-Type, +Type)
% subtypes van een supertype kun je bepalen door dat het subtype een implements of een extends van het supertype doet, afhankelijk van een class of interface als supertype
% de subtype_recursive geeft ook de input als resultaat terug in een findall
% de proper_subtype_recursive ALLEEN de subtypes in een findall
% findall(A,  ( member(Cls, Superclasses), subtype_recursive(A,Cls)), Z).
% findall(A,  ( member(Cls, Superclasses), proper_subtype_recursive(A,Cls)), Z).
subtype_recursive(Type, Type) :-
  classT(Type, _, _, _, _).

subtype_recursive(SubType, SuperType) :-
    proper_subtype_recursive(SubType, SuperType).

% zie commmentaar van subtype_recursive
proper_subtype_recursive(SubType, SuperType) :-
    direct_subtype(SubType, SuperType).

%
proper_subtype_recursive(SubType, SuperType) :-
  direct_subtype(SubType, IntermediateType),
  proper_subtype_recursive(IntermediateType, SuperType).

% implement van een interface of extends van een class
direct_subtype(SubType, SuperType) :-
  implements_directly(SubType, SuperType).

direct_subtype(SubType, SuperType) :-
    extends_directly(SubType, SuperType).

% directe implements van een interface
implements_directly(Class, SuperInterface) :-
    implementsT(_, Class, SuperInterface).

%parameteriseerde directe interface variant
implements_directly(Class, SuperInterface) :-
    implementsT(_, Class, ParameterizedType),
    parameterizedTypeT(ParameterizedType, SuperInterface, _).

%directe extends van een class
extends_directly(SubClass, SuperClass) :-
    extendsT(_, SubClass, SuperClass).

% parameteriseerde directe class extends variant
extends_directly(SubClass, SuperClass) :-
    extendsT(_, SubClass, ParameterizedType),
    parameterizedTypeT(ParameterizedType, SuperClass, _).

% -extends_recursively(+SubClass, -SuperClass)
% find all superclasses of subclass
```

```
63 extends_recursively(SubClass, SuperClass) :-
64     nonvar(SubClass),    % SubClass is not a free var
65     extends_recursive_from_subclass(SubClass, SuperClass).
66
67 % -extends_recursively(-SubClass, +SuperClass)
68 % find all subclasses of superclass
69 extends_recursively(SubClass, SuperClass) :-
70     var(SubClass), nonvar(SuperClass),
71     extends_recursive_from_superclass(SubClass, SuperClass).
72
73 %% extends_recursively(?SubClass, ?SuperClass)
74 % find all extension pairs
75 % subclass gebonden dan find all superclasses of subclass
76 % superclass gebonden dan find all subclasses of superclass
77 extends_recursively(SubClass,SuperClass) :-
78     var(SubClass), var(SuperClass),
79     type(SubClass),
80     extends_recursive_from_subclass(SubClass, SuperClass).
81
82 /**
83  * extends_recursive_from_subclass(+SubClass,?SuperClass)
84  */
85 extends_recursive_from_subclass(SubClass,SuperClass) :-
86     extends_directly(SubClass, SuperClass).
87
88 extends_recursive_from_subclass(SubClass,SuperClass) :-
89     extends_directly(SubClass, MiddleClass),
90     not(MiddleClass==SuperClass),
91     extends_recursive_from_subclass(MiddleClass, SuperClass).
92
93
94 /**
95  * extends_recursive_from_superclass(?SubClass,+SuperClass)
96  */
97 extends_recursive_from_superclass(SubClass,SuperClass) :-
98   extends_directly(SubClass,SuperClass).
99
100 extends_recursive_from_superclass(SubClass,SuperClass) :-
101     extends_directly(MiddleClass,SuperClass),
102     extends_recursive_from_superclass(SubClass, MiddleClass).
103
104 %% abstract_mehod(+Method)
105 %  een methode is abstact middel de modifier bij een class of in het geval
       van een interface
106 %  waarbij de mehthod altijd abstract (en public) by default is
107 abstract_method(Method) :-
108     methodT(Method, _, _, _, _, _, _, _),
109     modifierT(_, Method, 'abstract').
110
111 abstract_method(Method) :-
112     methodT(Method, Interface, _, _, _, _, _, _),
113     interface(Interface),
114     not(modifierT(_, Method, 'abstract')).
115
116 abstract_type(Type) :-
117     modifierT(_, Type, abstract).
```

```
118
119 abstract_type(Type) :-
120     interface(Type).
121
122
123  %% method_is_in_type(?Method, ?Type)
124  % bevraging +Method, +Type of method in type zit met -Type in welke class
125  % bevraging -Method, +Type welke methods in class, bij -Type welke
       methods in welke classes
126  % combi -Method, -Type onzinnig in gebruik = geef alle methodes in alle
       classes
127  method_is_in_type(Method, Type) :-
128   methodT(Method, Type, _, _, _, _, _, _).
129
130
131 %% same_signature(+Method1, +Method2)
132 % controleert of de naam en parameters voor override overeenkomstig zijn
133 % voor methods of bij constructor toepassing
134 same_signature(MethodID, AnotherMethodID) :-
135     methodT(MethodID, _, MethodName, Parameters, _, _, _),
136     methodT(AnotherMethodID, _, MethodName, MoreParameters, _, _, _),
137     not( MethodID = AnotherMethodID ),
138     equal_parameter_types(Parameters, MoreParameters).
139
140 same_signature(MethodID, AnotherMethodID) :-
141     constructorT(MethodID,_,Parameters,_,_),
142     constructorT(AnotherMethodID,_,MoreParameters,_,_),
143     not( MethodID = AnotherMethodID ),
144     equal_parameter_types(Parameters, MoreParameters).
145
146 %% same_signature_scoped(+Method1, +Method2, +Package)
147 % same_signatuur binnen de scope van een Package
148 % voor methods of bij constructor toepassing
149 same_signature(MethodID, AnotherMethodID, PackageId) :-
150      same_signature(MethodID, AnotherMethodID),
151      encl_package(AnotherMethodID, PackageId).
152
153
154 %% method_implements_abstract_method(+Implementatie, -Declaratie)
155 % er is sprake van implementatie van een abstract method als
156 % de methode zelf niet abstracts is, waarbij bovenliggende (transitieve)
       superclass
157 % juist wel de method als abstract gedeclareerd wordt.
158 % Voorwaarde van implementatie is net als bij override dat signatuur
       identiek aan elkaar is
159 method_implements_abstract_method(MethodImplementation, MethodDeclaration)
      :-
160     method_is_in_type(MethodImplementation, Class),
161     classT(Class, _, _, _, _),
162     not(abstract_method(MethodImplementation)),
163     extends_recursively(Class, SuperClass),
164     type_contains_method(SuperClass, MethodDeclaration),
165     abstract_method(MethodDeclaration),
166     same_signature(MethodImplementation, MethodDeclaration).
167
168 % geindexeerde aanroep voor bepaling of methode in Type (Class) zit
```

```
169 type_contains_method(Type, Method) :-
170    ri_methodT_parent(Type, Method).
171
172 % parameters zijn gelijk aan elkaar als type en aantal overeenkomstig zijn
173 equal_parameter_types([],[]).
174
175 equal_parameter_types([H1|T1],[H2|T2]) :-
176     paramT(H1, _, Type, _),
177     paramT(H2, _, Type, _),
178   equal_parameter_types(T1,T2).
179
180
181 equal_parameters([],[]).
182
183 equal_parameters([H1|T1], [H2|T2]):-
184     paramT(H1, _, Type, Name),
185     paramT(H2, _, Type, Name),
186   equal_parameters(T1,T2).
```

Listing C.3: inheritance

D

Exploring Refactoring

In this subsection, we position what we regard as refactoring, before we can address the refactoring related issues in more detail further on.

D.1. Refactoring and behavior

Often novice developers underestimate the complexity of their code and most of the time they try to refactor and build new functionality at once. Occasionally after several refactoring attempts, the novice developer manages to settle the refactoring without compilation errors and reasonable without failing unit tests. During refactoring, semantics comes in play as well. The *semantics* of the program (that is the expected functional behavior) after the refactoring should remain the same as before the refactoring. Then again, this may be not always obvious to the untrained student. Because even the tiniest alterations of code may break the expected behavior. To illustrate this, we will give an example of an ill-performed extract method refactoring that produces different outcomes before and after the refactoring steps.

Consider the following C# pseudo example [Verbaere et al., 2006] that clarifies why semantic preserving refactoring is not as trivial as it seems.

The penalty for making mistakes already starts in the original code. At line 3 we see variable 'i' defined, but depending on the flow of execution will or won't be assigned an initial value. Before line 8 variable 'i' has no value. Luckily, in this example, the code always reaches line 8. Modern state compilers might warn us about uninitialized variables. Some languages like C# enforces compile-time name binding.

The problem in the case of the extraction is that variable 'i' is returned (explicit *return* statement at line 16) without necessarily being assigned, because the assignment at line 13 depends on the execution of the 'if' statement at line 12. Proper data flow analysis again (*liveness* of variable 'i') on the extracted block hints that variable 'i' gets not assigned in case boolean value variable 'b' is false.

The following code snippets show what might be done to correct this. The variable 'i' at line 16 can safely be disregarded, since it has nothing to do with the variable 'i' at line 3. We can omit the function from returning the value of 'i'.

Table D.1: Refactoring learning - example mistake

Original code	Extract Method Refactoring

1 public void F(bool b)
2 {
3 int i;
4 if (b) {
5 i = 0;
6 Console.WriteLine(i);
7 }
8 i = 1;
9 Console.WriteLine(i);
10 }
	1 public void F(bool b)
	2 {
	3 int i;
	4 i = NewMethod(b);
	5 i = 1;
	6 Console.WriteLine(i);
	7 }
	8
	9 private static int NewMethod(bool b)
	10 {
	11 int i;
	12 if (b) {
	13 i = 0;
	14 Console.WriteLine(i);
	15 }
	16 return i;
	17 }

Table D.2: Refactoring learning - good refactoring example

Extract Method Refactoring done wrong	Extract Method Refactoring done right

1 public void F(bool b)
2 {
3 int i;
4 i = NewMethod(b);
5 i = 1;
6 Console.WriteLine(i);
7 }
8
9 private static int NewMethod(bool b)
10 {
11 int i;
12 if (b) {
13 i = 0;
14 Console.WriteLine(i);
15 }
16 return i;
17 }
	1 public void F(bool b)
	2 {
	3 int i;
	4 NewMethod(b);
	5 i = 1;
	6 Console.WriteLine(i);
	7 }
	8
	9 private void NewMethod(bool b)
	10 {
	11 int i;
	12 if (b) {
	13 i = 0;
	14 Console.WriteLine(i);
	15 }
	16 }

D.2. Stepwise mechanics, exercise

We will demonstrate an example Fowler [Fowler, 2018] gives how to apply to described mechanics in case of an Extracting Function refactoring, where the selected fragment of code contains local variables reassigned both outside and inside the newly extracted function. The provided source-code examples are Javascript based.

Extract Function mechanics:

1. Create a new function, and name it after the intent of the function
2. Copy the extracted code from the source function into the new target function.
3. Scan the extracted code for references to any variables that are local in scope to the source function and will not be in scope for the extracted function; pass them as parameters (a). If a variable is only used inside the extracted code but is declared outside, move the declaration into the extracted code (b). In the awkward case where the variable is used outside the extracted function. In that case, I need to return the new value (c)
4. Compile after all variables are dealt with.
5. Replace the extracted code in the source function with a call to the target function.

In this code excerpt below we want to extract the calculation for the outstanding variable to promote as a function on its own.

```
function printOwing(invoice) {
   let outstanding = 0;
   printBanner();
   // calculate outstanding
   for (const o of invoice.orders) {
       outstanding += o.amount;
   }
   recordDueDate(invoice);
   printDetails(invoice, outstanding);
}
```

The code to be extracted in our example is:

```
for (const o of invoice.orders) {
   outstanding += o.amount;
}
```

Because we modify (and therefore use) the variable outstanding within the code fragment to be extracted Fowler advises to bring together the definition of variable outstanding as close as possible to the use of variable outstanding. Fowler refers to the deployment of the Slide Statement refactoring. The code except now resembles code fragment, ready to be refactored further:

```
function printOwing(invoice) {
   printBanner();
   let outstanding = 0;
   for (const o of invoice.orders) {
```

```
5       outstanding += o.amount;
6    }
7    recordDueDate(invoice);
8    printDetails(invoice, outstanding);
9 }
```

Below is the situation after applying steps 1, 2 en 3b from the refactoring mechanics:

Table D.3: Extract method steps 1,2, 3b

Mechanic step	CODE
	1 function printOwing(invoice) { 2 printBanner(); 3 4 let outstanding = 0 5 recordDueDate(invoice); 6 printDetails(invoice, outstanding); 7 } 8
1 Creating a new function ->	9 function calculateOutstanding(invoice) {
3b Move into extracted ->	10 let outstanding = 0;
2\ Copy the extracted code ->	11 for (const o of invoice.orders) {
2/->	12 outstanding += o.amount; 13 } 14 return outstanding; 15 }

Proceeded by the applicable steps 3c and 5, when the code compiles successfully:

We tidy up the code even further when we declare the variable outstanding at line 4 to become a constant, since no reassignments take place any more. Another application that makes sense is to rename[1] the return value at line 14 to 'result'.

```
1  function printOwing(invoice) {
2     printBanner();
3
4     const outstanding = calculateOutstanding(invoice);
5     recordDueDate(invoice);
6     printDetails(invoice, outstanding);
7  }
8
9  function calculateOutstanding(invoice) {
10    let result = 0;
11    for (const o of invoice.orders) {
12        result += o.amount;
13    }
```

[1]The definition of outstanding at line 10 is not bound to the definition at line 4 (free local variable that shadows the global variable) so in any case we may simply rename it.

Table D.4: Extract method steps 3c and 5

Mechanic step	CODE
	1 function printOwing(invoice) {
	2 printBanner();
	3
5 Replace extracted code with call ->	4 let outstanding = calculateOutstanding(invoice);
	5 recordDueDate(invoice);
	6 printDetails(invoice, outstanding);
	7 }
	8
	9 function calculateOutstanding(invoice) {
	10 let outstanding = 0;
	11 for (const o of invoice.orders) {
	12 outstanding += o.amount;
	13 }
3c Var is used outside (@6)	14 return outstanding;
	15 }

```
14    return result;
15 }
```

E

Refactor Guidance intro

E.1. Generated advice intro

Based on the surveys conducted by Patrick de Beer in his thesis about Refactoring Guidance [de Beer, 2019], students get a better understanding of the refactor mechanics when they receive meaningful advice about the refactoring steps they should perform on a piece of selected sample code. Collaterally, it helps the student to learn about the code structure and underlying design.

The student picks from the set of currently supported refactor methods (rename method and extract method) one that should be applied. The tooling then walks though a graph containing predefined context detectors.

- Each detector that matches a certain precondition is responsible for part of the advice.
- Each detector also scans for characteristics specifically appointed to that detector.
- When fired the detector will serve a piece of template advice that will be enriched with actual class -, method - and field names based on the code under investigation.

The nature of the code context determines what the concrete advice will be, for example, when there is a method with public access that the advice will produce particular information concerning the effects public methods have on a renaming action. In general, the set of instructions addresses java constructs and could give students a better understanding of the underlying software design.

Following is a fragment of the advice given for the Rename Method refactoring, unleashed on following demo code.

```
1 interface API_Interface {
2
3     public void subscribe();
4     public java.lang.String getAccountName();
```

```
}

public class API_Implementation implements API_Interface {

    @Override
    public void subscribe() {

    }

    public java.lang.String getAccountName(String prefix)
    {
        return prefix + getAccountName();
    }

    public java.lang.String getAccountName() {
        return null;
    }
}


public class API_SpecialImplementation extends API_Implementation {

    @Override
    public void subscribe() {

    }

    public java.lang.String getAccountName()
    { // API_SpecialImplementation,  Rename line 34

        String tempString = "";
        tempString = "Hello";

        tempString.toUpperCase();

        return tempString;
    }
}
```

The next figure Figure E.1 is a screenshot of the actual prototype generation of advice for a Rename Method refactoring. It constitutes the identified risks associated with a Rename Method refactoring, tips in general as well as the given contextual instructions on the listed code, if case *getAccountName()* gets renamed to *HelloWorld()*.

The output may comprise:

- refactor generic risk based advice(s)
- detailed risk based advice(s) (based on detected language construct)
- refactor generic instruction advice(s)
- detailed instruction typed advice(s) (based on detected language construct).

The generated output represents a set of advices. Advices can be of type 'General instructions', 'Recommendations' or ' Warnings'. Each piece of advice is based on

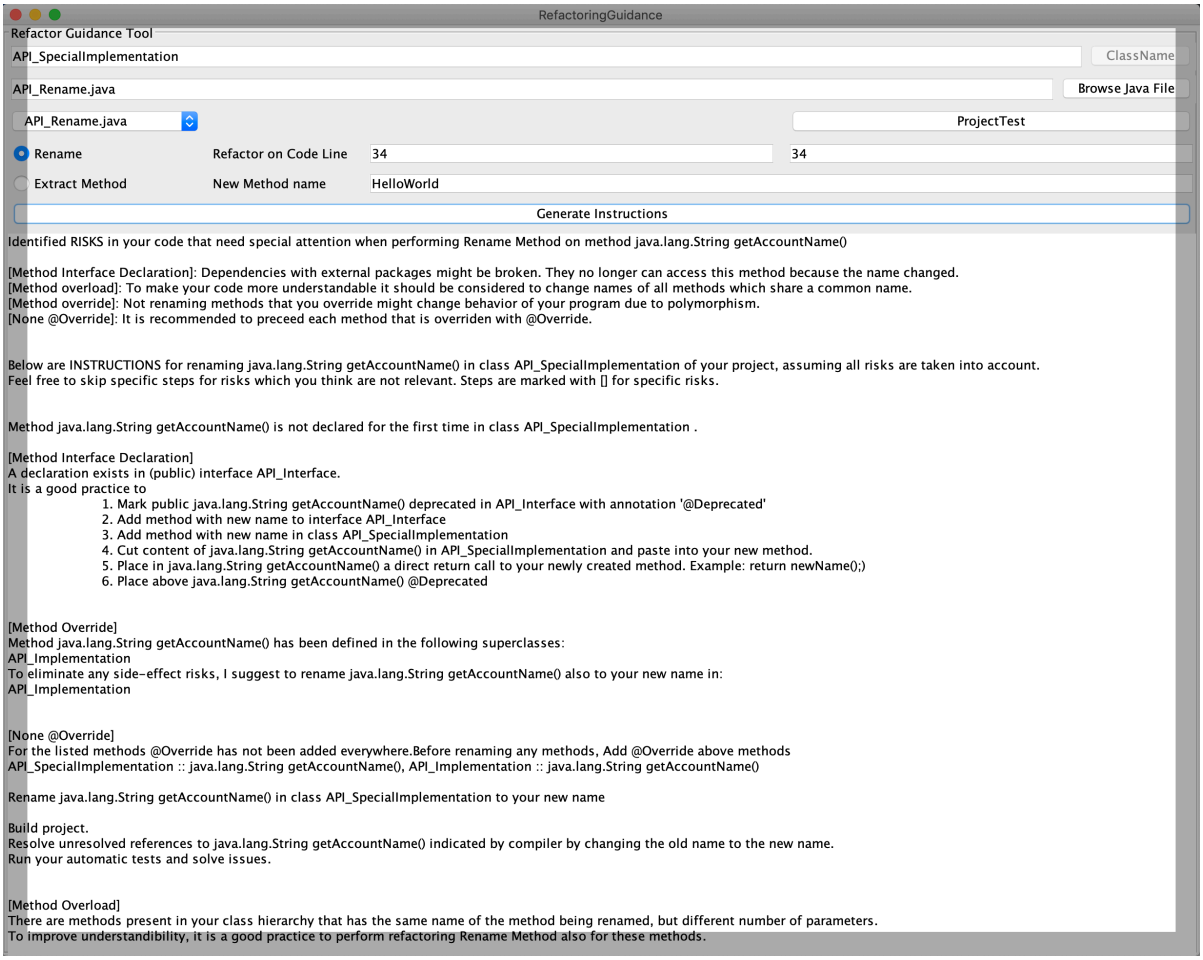


Figure E.1: Prototype rename method output

an advice template. The first line, for example, is a generic advice for the Rename method refactoring.
The tool is using an advice template for each given advice. What we see at the first line is that template method name has been substituted by the inspected method name. So, you do see the actual name, in this example the method name *getAccountName()*.

The tooling figures out that affected method *getAccountName()* is also declared in the superclass in a polymorphic manner. The student is neatly pointed to the implications for Java's method overloading and method overriding. For this to do, the tool dissects the code into compilation units, packages, classes, methods and fields. It can tell that the *getAccountName()* method is belonging to the API_SpecialImplementation class and is an override to the API_Implementation superclass. The method also gets overloaded, we have the *getAccountName()* and *getAccountName(String)*. And the tooling is aware that *getAccountName()* gets called in *getAccountName(String)*, so the referring call should be renamed as well.

E.2. Prototype inner details

Code Context Properties & - Detectors To give applicable advice to the student we need to analyze source code for certain code constructs that can be associated with some advice. Patrick de Beer [**?**] nominates these code constructs as "Code Context Property (CCP)". In order to detect such CPPs Patrick coins the term Code Context Code Property Detectors (CCPD)s. A detector (alias Context Code Property Detector) is bound to one specific property, so this means that one CCPD detector only looks for the characteristics of how to detect one specific CCP property.

RAG template example Figure E.2

Advice Templates The code context and in particular the selected code fragment (mentioned in the first paragraph above) will be input for the detector. In order to generate advice, the tool asks the detector if the property has been detected. If the detector positively finds its associated code context property, then the tool obtains a template advice.
Each template advice needs to be instantiated with the current values found in the code context. An advice template is generic for the code construct without actual values for names of classes, methods, fields and variables. These values can of course differ based on the actual code in question.

E.3. RAG collection

The Refactor Guidance prototype provides the RAGs for the following two refactorings: Rename Method Figure E.3 and Extract Method Figure E.4. Indicated with yellow surrounded boxes are the empty nodes. Empty nodes are artificial advice nodes purely results in sort of ANDing two or more CCA functions together without advice.

The Rename Method refactoring implementation, for example, currently involves the following recognized properties to detect if the method: is declared in a single class, declared multiple times (with the same signature) within its class hierarchy, declared as public interface, overrides its superclass method, is polymorphic (overloading

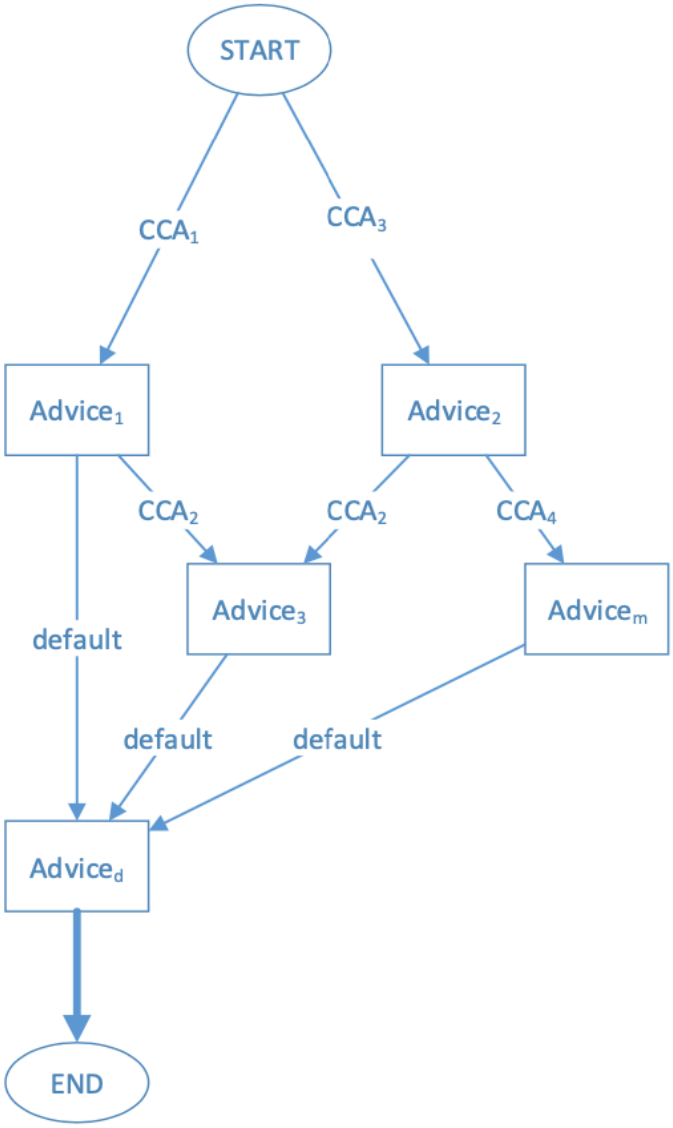


Figure E.2: Refactoring Advice Graph example

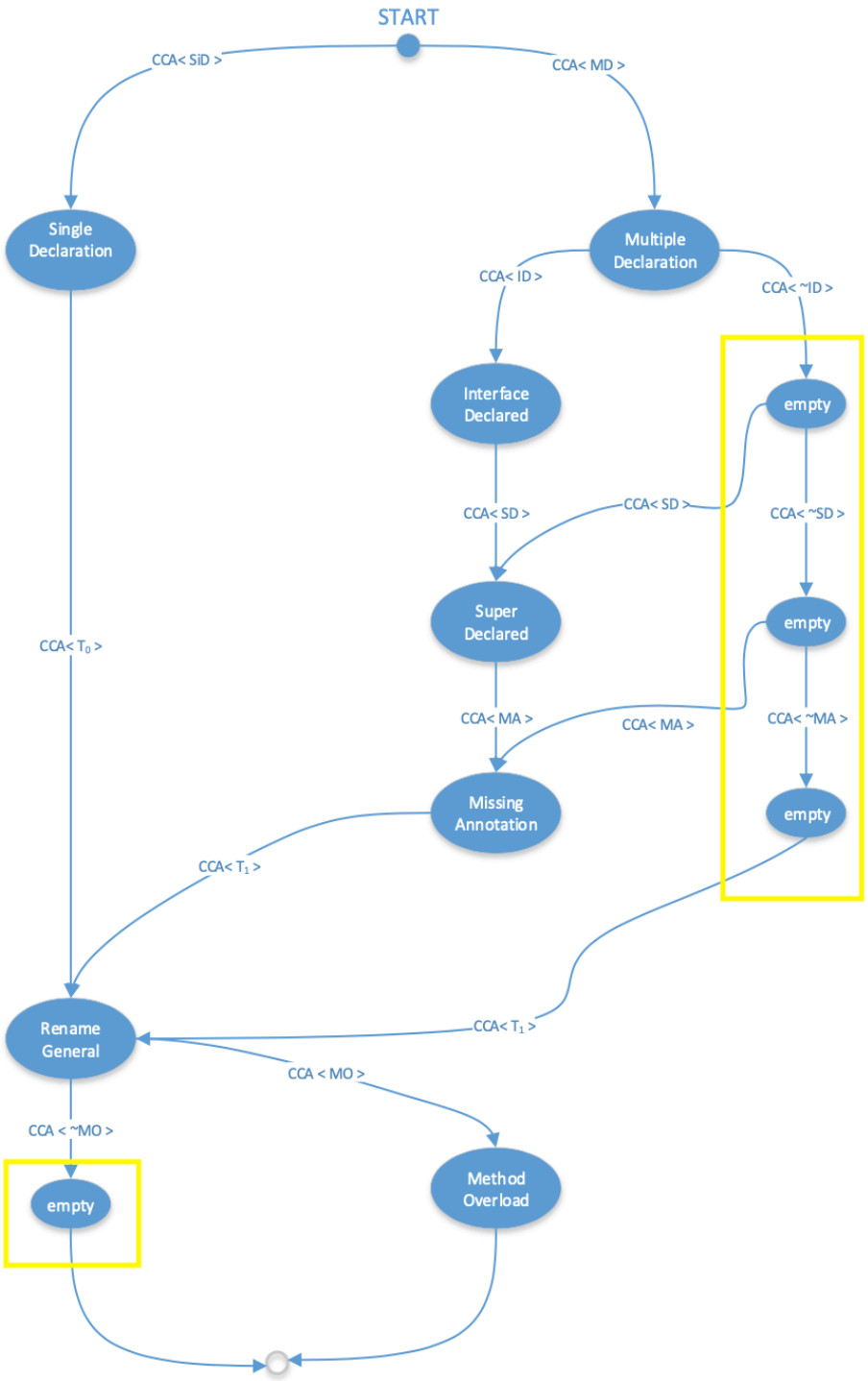


Figure E.3: Rename method RAG

methods in the same class but other signature) or has the @override annotation.

Detectors are associated with functions in code, so-called CCPD functions. In the same order as above for the Rename refactoring we have following property representing CCPD functions:

- SID (Singled declaration of method),
- MD (Multiple declaration of same signature method),
- ID (Method declared in public Interface),
- SD (Method declared in superclass),
- MO (Method is overloaded),
- MA (Missing override annotation).

The functions listed above are presented as edges.

The CCA functions will always fire 'true'. So this means the path will be traversed without detector triggering. The Extract Method RAG also has yellow rectangles around Complex *n* kind of advice. There is not a straight refactoring solution in these cases.

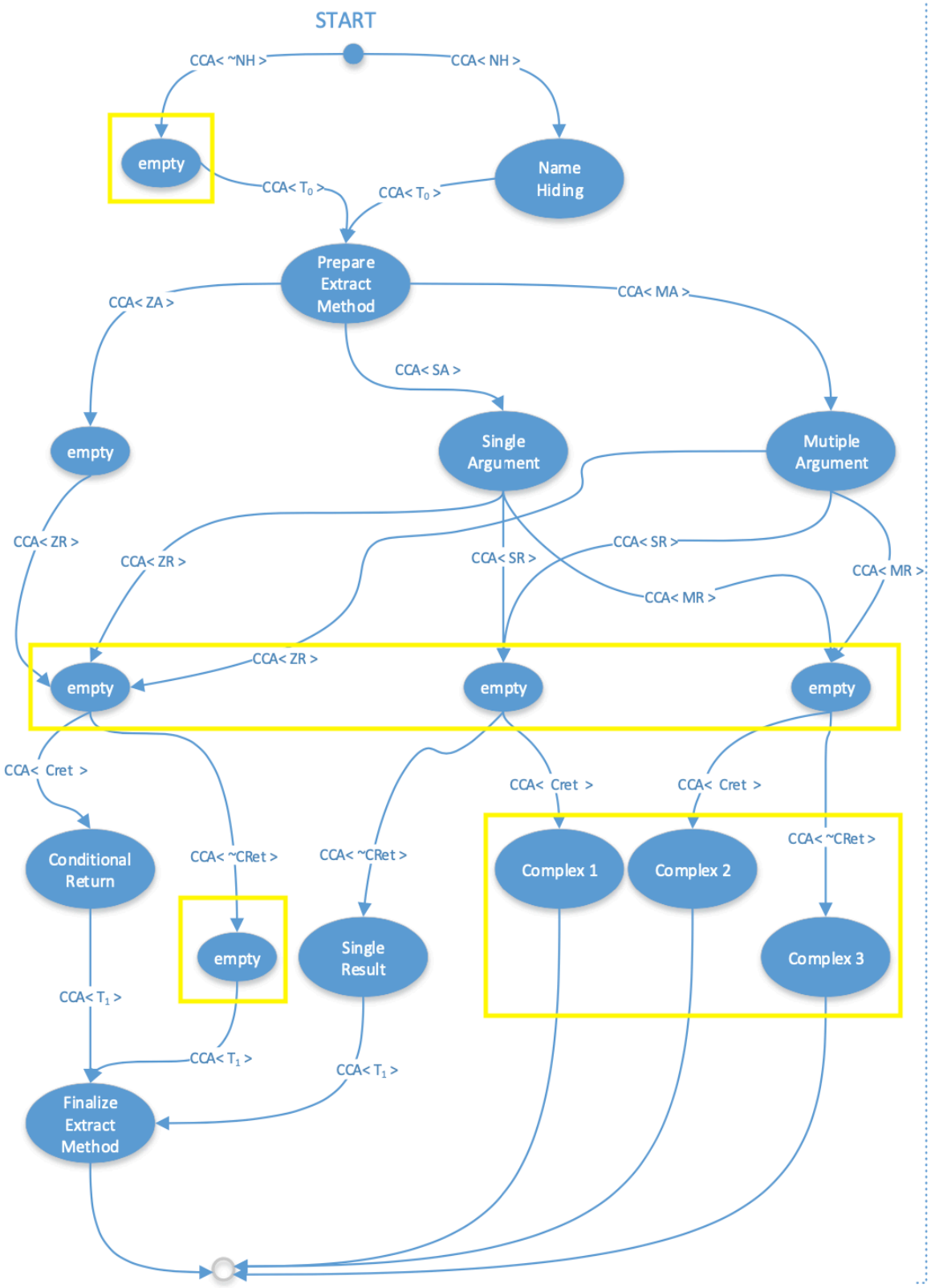


Figure E.4: Extract Method RAG